UNIVERSITY OF OSLO
Department of Informatics

Evolving Locomotion for a Quadruped Robot using Cartesian Genetic Programming and Physics Simulation

Master thesis

Anne Sigrid Aarstad

Spring 2012

Abstract

In this work I have explored and seen the effects of using Cartesian Genetic Programming to control a simulated quadruped robot. The control system is tested and explored using a variety of node functions, a larger and smaller number of function nodes, changes in physics settings, and a handful of different obstacles and sensors. Some interesting solutions were found, such as somewhat successfully climbing stairs. I also found a handful of very promising walking gaits, also some that are similar to earlier results using the same robot but with a different control system. However, I found that using such a general system might not the most appropriate for this rather limiting robot design. The added complexity gained from using this system made the search difficult, and reduced the chances of getting interesting designs. While the experiment setup would need further fine tuning I can recommend this system for use as a robot controller, though it would be more suited for a more general robot setup, or even an evolved robot morphology.

Contents

List of Figures

List of Tables

Preface

I wish to thank Kyrre Glette, my reasearch advisor, at the Institute of Informatics at the University of Oslo. I also wish to thank Simon McCallum for many good suggestions and help, as well as family and friends for help and support.

Chapter 1

Introduction

For a human engineer to program and develop a complex system it would require much of both time and resources. The development of complex behavior for a robot, for instance to operate in unorderly or undefined environments – environments not previously seen by the robot or the engineer that built it, is a task that is continually being researched. An example of an instance where the ability to operate in such environments could be useful, is in finding survivors or radioactive leaks in a disaster area where it would be dangerous to send in humans. At present, these robots would have to be remotely controlled by humans, which makes it unlikely that very many can be controlled at once. Instead, it would be desirable to have a large number of robots to collaborate autonomously with mapping the area.

One branch of research toward such robots is evolutionary robotics. Here we wish to automatically, without much human intervention let robots program themselves. This process is inspired by Darwin's "survival of the fittest". It consists of generating an amount - a population - of random programs. The ones who perform the task better are selected, and these are combined ("mated"), and a new population containing the new individuals (their "offspring") will continue the cycle until we have a program that performs the chosen task sufficiently well.

For this master thesis I will describe and develop a system to perform this type of evolution of programs, in particular towards robots in complex environments, and perform experiments using this system. This system expands and builds upon the system shown in [1], built for use by the ROBIN (Robotics and Intelligent systems) group at the Department of Informatics at the University of Oslo.

1.1 Goals of the thesis

The research goal of this thesis is to find interesting robot behaviors to evolve and perhaps use in real robots. Towards that goal I wish to investigate what effect controlling the joint angles (of a robot) directly based on sensor input, through a Cartesian Genetic Programming graph, will have on locomotion in a complex environment.

The development goal of this thesis is to perform experiments toward interesting robot behavior, and towards that goal expand upon and add features to

an existing evolutionary robotics platform in development at the ROBIN group.

1.2 Outline

The thesis is organized as follows: The Background chapter will first give an overview over evolutionary algorithms, before giving a more detailed description of the variant Cartesian Genetic Programming. Lastly I will describe some relevant additions for evolutionary robotics. In the Implementation chapter I will describe the robot used, its control system, and the experimental setup. In the Experiments chapter the different experiment sets will be described, as well as a short discussion of the results. In the Discussion chapter the most noticeable features of the results will be discussed further, some possibilities for future work will be mentioned, and then the final conclusion.

Chapter 2

Background

The works I will build my thesis on are Karl Sims works on virtual creatures [2, 3], Lassabe's expansions on these [4], and The GOLEM Project (Genetically Organized Lifelike Electro Mechanics) [5], that deals with prototyping this kind of robots in the real world. For sensor modeling and evolving sensor morphology I have taken inspiration from France in [6], Parkers work in [7, 8, 9] and from Balakrishnan in [10]. For the representation of the control system I will utilize Millers "Cartesian Genetic Programming" [11], and inspiration from Harding and Miller in [12] where a robot controller is evolved using this approach. For an overview into the field, and for inspiration towards the control system I have had great use of Nelson [13] and Sprong [14], as well as Poli [15] for an overview on genetic programming.

2.1 Evolutionary Algorithms

In this section I will summarize the components of evolutionary algorithms, in particular from the perspective of evolving and simulating virtual creatures, for the purpose of prototyping them in the real world.

First a small note on my naming conventions: There are four different dialects of evolutionary algorithms, under the common name "evolutionary computing". These are: genetic algorithms, genetic programming, evolutionary programming and evolution strategies. I will not consider their specific differences here. I will describe the concepts in more generic terms and as a simplification call them evolutionary algorithms.

Evolutionary algorithms are a method of programming computers that is inspired by natural selection. Instead of programming a solution directly we generate a large number of random solutions, and select the best among them to keep before generating a new 'population'. Also inspired by natural selection is that we have mutations on our solutions, and that we 'mate' them, that is we use parts of two solutions to create a new one. The 'best' solution is measured by the fitness (or objective) function of our algorithm.

As an overview over what parts an evolutionary algorithm is composed of, and how it normally runs, I will briefly explain each step in the flow chart in

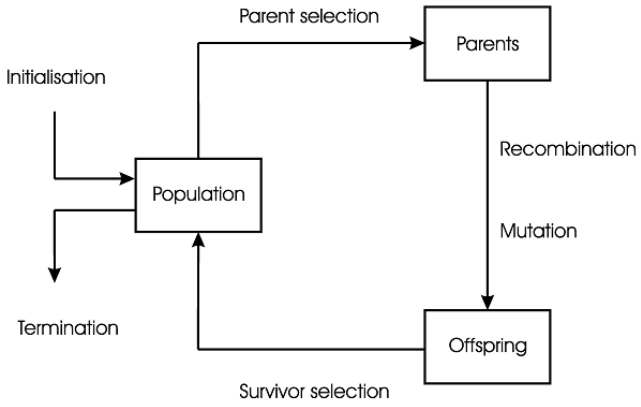

Figure 2.1: The steps in an evolutionary algorithm, from [16].

figure 2.1. I will also mention which chapter describes them.

A **population** of solutions (candidate programs) is initialized randomly according to the **representation**. All the individuals in the population is evaluated by a **selection criteria** (or fitness function), and each individual given a fitness value. **Parents** are selected: The individuals with high fitness values are preferred for selection, but those with low fitness values are given a small chance as well. **Variation operators** are applied, producing new individuals called the **offspring**. To keep the population size stable, **survivors** must be selected. Often either there will be only offspring and entirely new random generated individuals, or the parents are kept and the individuals with the lowest fitness removed. This new population is evaluated again and this loop continues until it reaches a **termination** condition. This can be, for instance, the number of generations run, or that there has been no change in fitness over a given amount of time.

2.1.1 Representation

An important choice to make when designing an evolutionary algorithm is the choice of how to represent the problem internally in the computer. This representation, that all the calculation will be performed on, is often called the genotype (of the solution). When a solution is converted back to the actual problem (so that it can be evaluated for fitness) it is called the phenotype. Often, these are completely different and separate, but not always. For some problems it is appropriate to do the calculations directly on the phenotype. In figure 2.4 on page 15 we can see an example of a genotype and it's resulting phenotype.

Most often the representation is the genotype and is evolved on. It is then interpreted into the phenotype to be evaluated. Sometimes the representation is separate from the genotype, as in many cases with binary string representation, and then the genotype is only a step in the interpretation towards the phenotype. For certain problems a separate step of 'repairing' the phenotype is required in those cases where there are requirements on the shape of the solutions, and a

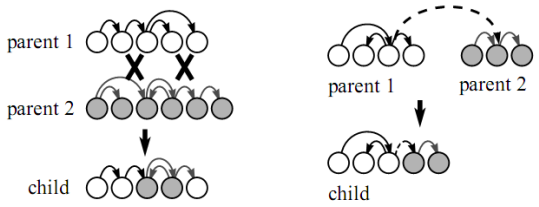

Figure 2.2: Mating directed graphs, from Sims [2].

direct interpretation of the genotype would lead to an invalid solution.

One type of internal representation that is commonly used are strings (arrays) of numeric values. Binary, integer, real numbers or floating point values are often used. Sometimes using a tree or a graph of nodes fits the problem better. Most of the works I will build on have utilized graphs in their solutions. A tree or graph might be used to code for mathematical expressions, logical expressions, or even program code. It always depends on the problem.

2.1.2 Variation

In an evolutionary algorithm, the variation is the most important part. How it will be implemented depends on the representation of our problem. But it will in most implementations include a type of recombination, and some form of mutation. Recombination is modeled after mating in biology. Mutation is a random change to some value in the representation. Both these are called variation operators.

Mutation is most of the time a very simple operator. It takes a single individual as input, and produces an individual similar to it with some small change. If the representation is a string of binary numbers, we will simply flip a random value from 1 to 0 or 0 to 1. For representations with for instance floating-point or integer values one can add, subtract or even replace a number with a new number. For trees and other more advanced structures mutation can also be used. Then it is appropriate to add or remove an element, or perhaps randomly move an element to a different position.

Recombination is an operator that takes two (or more) individuals, and combines them in some way to produce one or more new individuals with features from both. For representations with strings of values we copy values from the first parent up to a randomly selected point, and then continue copying from the other parent. This also depends on the implementation. If there can be only one of each value (for instance when evolving a specific order of actions), more care must be taken. The approach is mostly the same for trees and other more advanced structures.

Sims uses directed graphs in his work. Mutation and recombination on graphs is much the same as with trees. For mutation nodes are randomly added or removed, the same with the connections. Or small alterations are made to the

internal parameters of the nodes. Recombination is started by copying part of one graph and then combining it with part of another graph, see figure 2.2 on the preceding page. Lassabe's work uses much the same techniques as Sims, and also moves the links between nodes.

2.1.3 Selection Criteria

The selection criteria or fitness function is a way to measure the performance of the individuals. The individuals are given a fitness value; the higher the value, the better the individual solves the problem. What form the fitness function takes depends on the problem. In this case the evaluation is performed using a physics simulation in a 3D environment, measuring distances traveled in this virtual space.

Sims, Lassabe and the Golem Project use similar measurements. For walking they use either distance traveled, or distance traveled per unit of time. For jumping behaviors (not implemented in the Golem Project) they use the height above ground for the lowest part of the creature.

2.2 Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) is a variant of genetic programming that represents a graph as a string of integers coding for the connections and functions of the nodes. Normally in genetic programming the program is represented as a tree. 'Cartesian' refers to how the graph is mapped. One of its interesting features is that parts of the graph might be disconnected from the rest of the graph, and not used. This becomes similar to 'junk DNA' ('introns'). By mutation connections can be activated or deactivated. The graph has a set number of inputs and a set number of outputs. The signal generally moves in one direction, from inputs to outputs. The user set variables are: number of rows, number of columns and how many levels back connections can be made. The most common is to have it be one or two columns back, although for some problems it is appropriate to let the entire graph be connectible, or only the nodes to the right if a feed-forward type of structure is desired. Each node has a number of inputs depending on the function it performs. There is a type of lookup table for the functions.

CGPs take the form of an indexed graph encoded in a linear string of integers. It most often represents directed acyclic graphs. The inputs and node outputs are numbered sequentially, while the node functions are separately numbered. The genotype is a list of node connections and functions. It is mapped to an indexed graph that can be executed as a program (the phenotype), an overview is shown in figure 2.3 on the next page. CGP is a general representation, it can represent neural networks, programs, circuits, and many other computational structures. In [17] they mention in addition uses in evolved art.

In table 2.1 on the facing page and figure 2.3 on the next page we can see the different variables used to define a CGP. The nodes in the same column are not allowed to be connected to each other. The most freely connected network

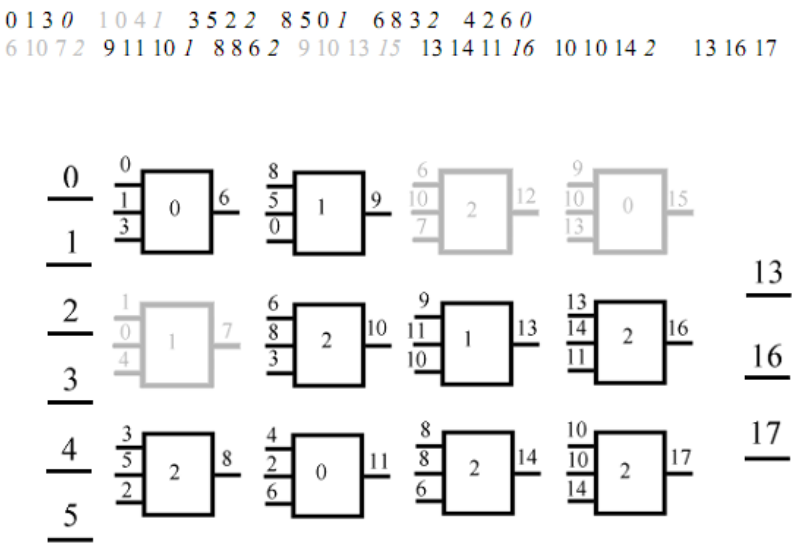

Figure 2.3: An overview of a CGP graph, from [11] At the top we have the chromosome, at the left we have the inputs and at the right we have the outputs. Each group of three digits in the chromosome codes for a node in the graph. The last digit in a node code is the function it performs, the rest are its inputs. The last group of digits are the outputs of the graph.

Table 2.1: CGP variables

`i`	inputs	set of program inputs
`o`	outputs	set of program outputs
`f`	functions	set of node functions
`n`	nodes	number of nodes: `r` × `c`
`r`	rows	number of rows
`c`	columns	number of columns
`l`	levels back	number of previous columns a node can connect to
`a`	arity	number of node inputs, the maximum arity of `f`

is when `r=l` and `l=c` so that any node can be connected to any on the right. The maximum length of the genotype is `r×c(a+1)+o`.

Mutation allows activation and deactivation of redundant code or 'junk DNA' nodes. A node is 'junk DNA' when it's output is not used by any other nodes. Most implementations use only mutation. Recombination in CGP usually would not create an individual with features of the parents, and often proves more disruptive than helpful (for more on this see [17] page 29).

2.2.1 Advantages and disadvantages

There are many advantages of CGP compared to other forms of genetic programming. Certain features can be both an advantage and a disadvantage, and I will mention these first.

It is possible for many genotypes to map to the same phenotype. This is the case when a change in the genotype does not cause a change in the graphs outputs. Such different genotypes with the same fitness, as well as the existence of 'junk DNA' allow for easily escaping local maxima. Sometimes this is a problem, as the search jumps away from many good solutions it finds as well as jumping into them.

Some of the advantages of CGP are that graphs are more general than trees. They can be applied to a greater variety of problems. The genotypes are fixed length, and the phenotype is of variable length depending on the number of unexpressed genes.

2.3 Evolutionary Robotics

As mentioned in the introduction there is much being developed in the field of evolutionary robotics. In this section I will mention some obstacle and sensor setups I am basing my work on. In addition I will mention Physics Simulation and 3D Printing in particular as these are a common addition to a system used for evolutionary robotics.

2.3.1 CGP robot controller

Miller and Harding wrote a paper [12] on evolving a robot controller using CGP. They used a CGP with 20 columns and 2 rows, with signed integer operations. Levels back was set to be the number of columns. The robot was a two-wheeled differential drive robot, with two distance sensors pointing forwards, separated by 20 degrees.

The mathematical functions used were:

- Add, subtract, multiply, divide: two inputs, integer arithmetic. Dividing by 0 returns 0.
- Compare: returns +1 if the first is greater, 0 if equal and -1 if less than the second input.

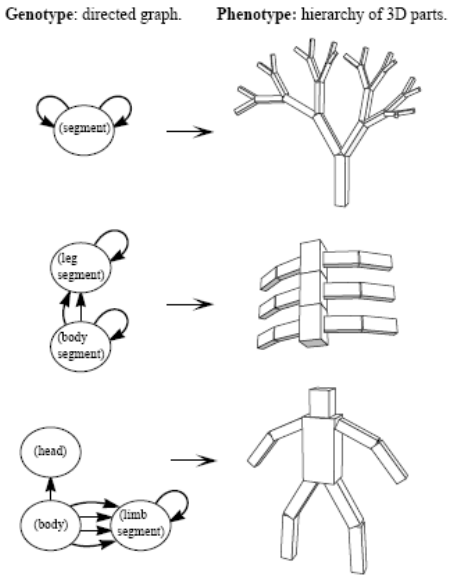

Figure 2.4: Some simplified hand-designed examples illustrating the relationship between the graph genotype and the creature morphology phenotypes used by Sims [2, 3].

- Min and max: returns the minimum of the maximum of the two inputs.
- Fixed integer: stores a value between -100 and +100. No inputs.
- Two output nodes. The integer value is found, truncated to between -100 and +100, and scaled to between -1 and +1.

Their evolutionary setup consisted of running with a population of 40 individuals, with a limit of 1000 generations, terminated at a max fitness. They used no crossover, and a mutation rate of 5% of the node count, tournament selection of size 5, and elitism of 5 best to next generation.

How this setup compares to mine will be discussed in section 3.3.1 on page 25.

2.3.2 Representation

Sims uses nested directed graphs for both morphology and behavior [2, 3]. He has one graph representing the structure, called the morphology, of the individual (in this case a virtual creature composed of blocks and joints). This graph codes for what parts the creature is composed of and how they are connected (figure 2.4). Each node of this graph has another graph connected to it, which codes the behavior of the individual. It represents how the individual receives sensor data, and controls the joint connected to the next virtual body part.

In the Golem Project [5] they represent their creatures using attributes with strings of integer and floating-point values. These code for the various properties

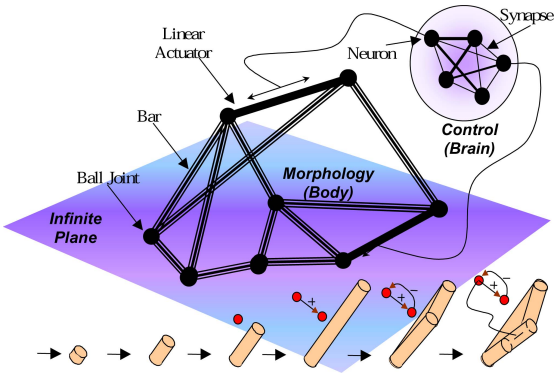

Figure 2.5: Illustration of a sample creature morphology used in the Golem Project [5].

of the bars, actuators and artificial neurons their creatures are composed of, along with the connections between them (figure 2.5).

Balakrishnan [10] represented both his sensor positions (2-tuples of i and j values) and neural network (connectivity matrix with weights) as a string of real values for their robot controller. The neural network consisted of input nodes from all the sensors, and two action nodes (outputs to the motors), one deciding move vs turn, the other the turn direction.

Parker [9] represented his control system as a string of bits, containing the various properties of his sensors (heading, range, and whether the sensor is active or not), and which of the 16 preset gaits to use.

2.3.3 Obstacles

Parkers obstacles in [9] were 30 by 30 cm square boxes 10 cm tall. Eight of them were placed randomly in a 3 by 3 meter walled testing area. The positions of the boxes were changed randomly by a slight amount for each run, and each robot/individual was run 3 times. This was to add some noise to the environment and so make the control systems more robust, and more suitable for reproduction in a real robot.

Miikkulainen had as one of his obstacle setups a series of low fences, arranged as square around the robot starting point. The aim was to evolve symmetrical gaits that could handle such hindrances well. He forces symmetry by grouping different pairs of legs to act in unison, forming gaits like trot, canter and bound. The reason he gives for this is that gaits evolved without symmetry often are ungainly and somewhat resembles crippled animals.

In his article [4] Lassabe describes his experiments on evolving virtual creatures in complex environments. He investigates several common complexities often encountered in real world environments. The real world is not a flat plane, there are gaps and stairs (figure 2.6 on the next page), and even more uneven

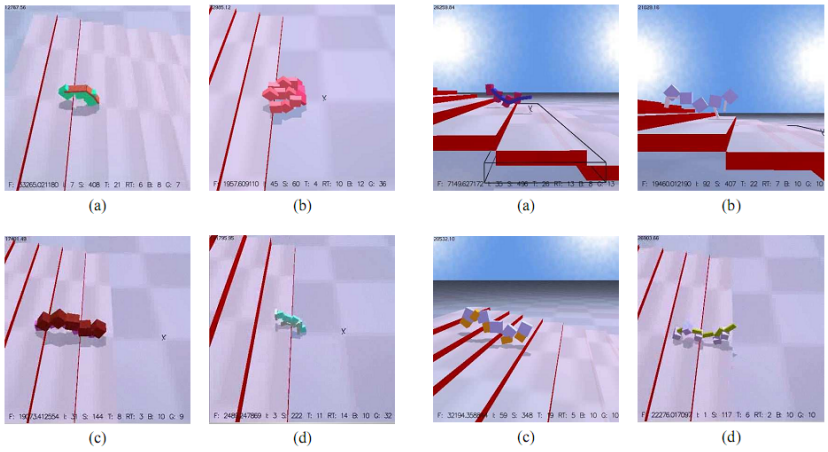

Figure 2.6: Various examples of stair-climbing robot, from Lassabe [4].

and difficult terrain. Robots that can traverse these would be much more useful in real world applications.

2.3.4 Sensors

Parkers sensors were arrayed on a platform mounted on top of the robot. He had four distance sensors and four tactile sensors. The positions of the sensors on the platform, as well as their direction and whether the sensor was active or not could be changed during evolution. The possibility of evolving the sensor positions will be further discussed in section 5.2.3 on page 52. In Miller and Hardings evolved robot controller they used two distance sensors pointing forwards and 20 degrees apart.

2.3.5 Physics Simulation

For some problems it might be appropriate to program a custom system for physics simulation, and sometimes an already existing library will do the job nicely, and possibly more reliably.

There are advantages and disadvantages to programming a custom built system. You can make the system do exactly what you want, but it is time consuming and frustrating work, especially when making such a system for use with evolutionary algorithms. Any errors in the system will be exploited by the evolving creatures, so as Sims mentioned in [2]: “Although this can be a lazy and often amusing approach for debugging a physical modeling system, it is not necessarily the most practical”. Both Sims and the Golem Project appear to have developed their own systems for simulating physics.

Using libraries has its own advantages and difficulties as well. It will do the calculations correctly, but your own program will need to be adapted to work with the library, and it might not have all the features you need. There are many libraries available that are appropriate, such as PhysX, Newton and

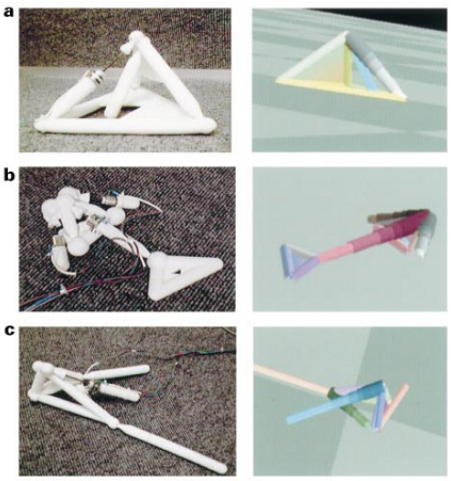

Figure 2.7: Robots reproduced in reality, from [5].

ODE. Lassabe used Breve, an engine based on ODE. He mentions that it is well suited for artificial life simulation, as it provides primitives, physics calculations and collision detection.

2.3.6 3D printing

The goal of the Golem Project was to create a system that could, without human intervention, automatically generate and prototype new robots (figure 2.7). Using what they called their 'solidifying technique' the points and lines representing an evolved robot are converted into more solid 3D structures with ball joints and accommodations for linear motors, and then printed on a 3D printer. Stepper motors were fitted in and connected to a microcontroller running the evolved behavior. They wanted the process to be as automated as possible, in order to explore the possibility of self-replicating robots.

2.3.7 System

The system used and expanded in this work is a system in ongoing development at the ROBIN (Robotics and Intelligent systems) group at the Department of Informatics at the University of Oslo, and previously shown in [1], and [18].

In this program the PhysX library is used for physics simulation, the GAlib (genetic algorithm library) is used for the evolution, and OpenGL, with GLFW and GLEW, is used for visualizing the robots in the simulated environment. An overview of the communication between these components is given in figure 2.8 on the facing page. A visualization of the simplified physics model blocks of the robot is shown in figure 2.9 on the next page.

The system is quite modular, making it simple to add new robots, new fitness calculations or other new features like sensors or entirely new control systems.

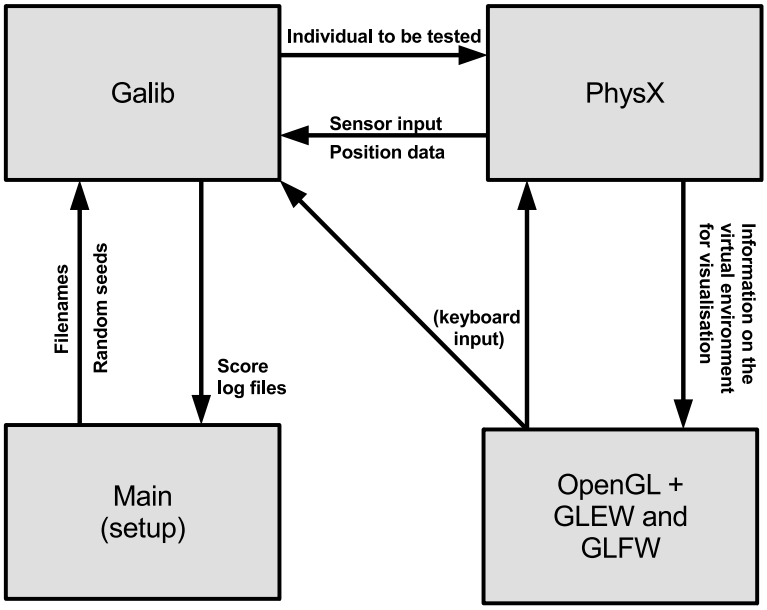

Figure 2.8: A simplified overview of the communication between PhysX, GAlib and OpenGL.

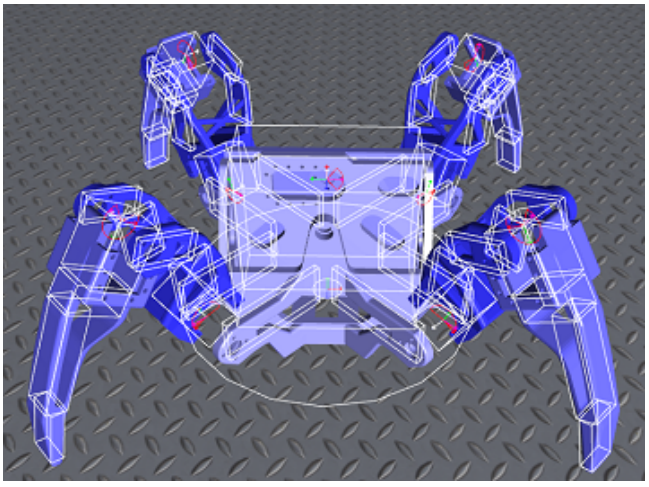

Figure 2.9: The simulated robot, here shown with the physics model in white, image from [18].

Through GAlib it is easy to change the evolutionary setup, from simple things like change the mutation rate to using entirely different representations.

Chapter 3

Implementation

In this section I will describe the system, and the methods I have employed. I will describe the various components of my system, the behavior I wish to investigate, and the approach I will use.

I will evolve robots that will traverse an environment, in a manner similar to Parker [7]. The control system will take inputs from various sensors, such as tactile sensors and light sensors, and output to joint motors on the robot. This will be simulated in a virtual environment, run using the PhysX physical simulation library.

The approach I will use for the evolution of the control system for my robot is a variant of Genetic Programming called Cartesian Genetic Programming (CGP), mentioned in 2.2 on page 12. The robot is a static robot, the possibility of evolving robot morphologies is also interesting, and will be explored later, as a possibility for future work (see section 5.2.2 on page 52).

The project was initially inspired by Sims work and has moved quite far from that starting point. The control system is inspired by Sims and Miller, and the sensors are inspired by Sims and Parker. The environments are inspired by Parker, Lassabes and Miikkulainens work. One of the main goals of the system is inspired by the Golem Project.

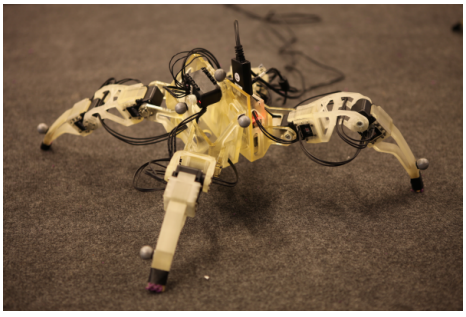

Figure 3.1: The four-legged robot we are evolving the control systems for, from the ROBIN-group.

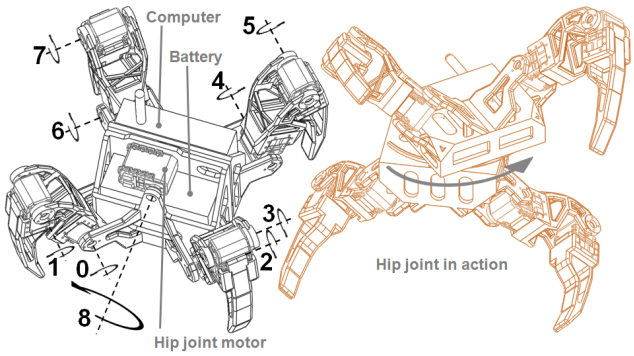

Figure 3.2: Diagram of the joints of the robot, and how they are limited. Images taken from [18].

3.1 Robot

The robot I am evolving locomotion for is an already existing robot developed at the Cornell Creative Machines Lab [19], and that is currently in use at the Robotics group at the Institute of Informatics at the University of Oslo (mentioned in [18] and [20]).

3.1.1 Morphology

The robot we generate the control systems for is a four-legged robot with 2 degrees of freedom (DOF) for each leg, as well as a 1 DOF central joint, see figure 3.1 on the preceding page. The joints of the robot are rather limited, see figure 3.2, and this limits greatly what manner of gaits we can achieve.

One possible expansion on this system is to generate the morphologies as well as the control system for the robots. This was explored in the Golem project [5], and will be discussed further in the Future Work section 5.2.2 on page 52.

3.1.2 Sensors

I have implemented three types of sensors for this system. These are: joint angle sensors, touch sensors and distance sensors. The values of the sensors are inserted into an array that is used as input for the CGP graph.

The joint angle sensors gets its values from PhysX using the `NxRevoluteJoint getAngle` function. There is one sensor per leg joint of the robot. There is no sensor for the central joint, but one could easily be added. The reason for this is that we are most interested in the values for the legs.

The touch sensors are set by a PhysX `trigger report`. Most of this was already implemented in the system. I placed one sensor on the tips of each of the legs of the robot. They give an output of 0 if not triggered and 1 when triggered.

The distance sensors utilize the PhysX raycasts. I placed two rays pointing forwards with an angle of 30 degrees between them. They give a value between 0 and 500, that is then scaled to between 0.00 float and 1.00 float.

3.1.3 Behavior

We are evolving gaits or forms of locomotion for this robot. The possible modes of movement are rather limited, as was shown in figure 3.2 on the preceding page. The joints are also limited in how far they can move and how fast they can move. From earlier work, using the parametrized control system [18], the best evolved gaits were walking somewhat like a turtle, twisting the central joint slightly and lifting diagonal legs symmetrically and alternating evenly between them. Most of the work would be done with two front legs, so it is indeed rather reminiscent of a turtle walking on land.

The possibility of goal homing and obstacle avoidance was also developed. For goal homing, the approach was to have a goal direction, and measure how far off from this direction the movement of the robot is. Some scoring would also be given for distance moved. This was never used in my sets of runs, but remain as a possible future direction of study. Several obstacle sets were developed, along with distance sensors, for the possibility of investigating obstacle avoidance, which are discussed in section 5.2 on page 52. No runs for obstacle avoidance were performed, however, many of the obstacle sets were still used, but not the distance sensors.

Having an evolved graph, like Sims, where both the connections and the functions of the nodes are evolved was long a goal of this system (this is described more in detail later). As I wanted to expand the previous simple parametrical system with sensors and attempt to make it more dynamic, the goal became to have a graph that performs mathematical functions on sensor input and that outputs this to the joint motors. When I learned of CGP, it seemed to fit what I wanted exactly.

3.2 Environment

The environment the robots will be tested in is most often a flat ground plane, but for some experiment runs there will be landscapes of obstacles. Most of the obstacle setups are inspired by the obstacles mentioned in section 2.3.3 on page 16. Many of the obstacle sets also create a sort of incremental fitness in that the robot often must learn to move along the flat ground plane before reaching the obstacles. Once it reaches the obstacles the challenge becomes a different one, even if the fitness calculations are still the same.

3.2.1 Obstacles

I designed several different obstacle setups. The most basic one is the simple ground plane. More complex ones are: Stairs, random room fixed for all robots, random room different for each robot, random boxes in a corridor and a hand designed pile of boxes at the end of a corridor (for climbing). The thought was

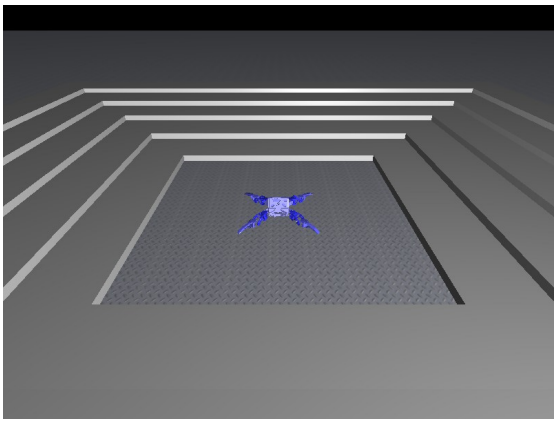

Figure 3.3: The simulated stair obstacle as it was used in the experiments, here with a unanimated robot for size comparison.

to have some obstacles that were too tall to climb over, and would have to be walked around, and some that were low enough, or started low enough to be possible to climb over.

The stairs are inspired by Miikkulainens fences (in [21]), and surround the robot starting point in a similar manner. The robot will have to learn to walk some distance before it encounters the stairs. The obstacle is shown in figure 3.3. The height of the steps is 2.5 cm and the depth of each step is 20 cm. There are five steps.

The random rooms are composed of randomly placed boxes with walls at 5 meters around the starting point. This is somewhat inspired by Parkers work in [7]. He also did some work on avoiding the reality gap, this I will discuss further in 5.2.4 on page 52. As we can see in figure 3.4 on the facing page, an area in the middle is left free of obstacles so that the robot does not immediately collide with them. The default is that the randomly placed boxes depend on the random seed of the entire run, but the possibility of having a different layout for each robot in a run was also implemented. A setup more like a corridor was also made, to encourage learning locomotion before reaching the obstacles.

A hand designed setup of boxes was also prepared, at the end of a walled corridor. This was designed with the possibility of learning climbing.

3.3 Evolutionary algorithm

In this section I will describe the evolutionary algorithm used, as well as give an overview of the control system.

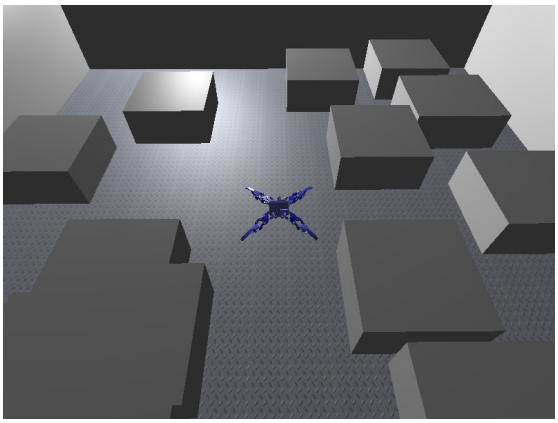

Figure 3.4: The simulated random boxes obstacle set as it was used in the experiments, here with a unanimated robot for size comparison.

3.3.1 Control System

The control system previously used in this system is a simple parametrized sine wave control system that sends a different signal to each joint, and each type of joint has different constraints. I wished to expand this system with something that has more variation in what functions is performed, and that also can use sensor inputs as part of the calculations. This idea was inspired by Sims control system in [2], where nodes of a wide variety of functions would be created as needed by mutations during the evolution, and the connections between them would also be changed during evolution. I originally had been sketching up my own system for this, but fairly soon I learned of Cartesian Genetic Programming (CGP) that seemed to do almost exactly what I had been planning to do.

Miller, the creator of CGP, had on his website a C implementation for CGP (a link to the website in in citation [22]). I could not use his code directly for my application, but many of the parts not directly associated with my application were inspired by his code, and resemble his in many ways.

I use GAlibs real numbered allele set genome for the array of integers. It allows setting ranges and steps for allowed values, and this can be different for each element in the array.

In my implementation of CGP I have eight inputs to the graph (possible sensor inputs), nine outputs (one for each joint motor), five rows and five columns of nodes, a node length of three and twenty different possible node functions (with maximum arity of two). I am not implementing levels back, as I am allowing the entire graph to connect to each other. Levels back would have been the entire graph, so I chose not to add that extra layer of complexity. This also means that rows and columns also become sort of irrelevant, as there is no distinction, so I might as well call it 25 nodes.

My implementation has several notable differences compared to the usual setup mentioned in 2.2 on page 12. My graph is cyclic, and only feed-forward in the sense that the graph has inputs and outputs. The nodes are allowed to connect to nodes in their own column, and to nodes both on the left and right. The reason I have chosen not to implement any limitation on this is that I wish to investigate a very open and free graph structure, that is more similar to Sims work that Millers in this respect. Because of these cycles I parse the graph twice, as some node inputs may not yet have given a value in the first pass due to being further along. The cycles are then never looped more than once. This is a rather experimental arrangement, and it might be one of the larger causes the difficult search space problem I will discuss in section 5.1.2 on page 51.

The inputs to the graph are set to `0.5` by default. In the experiments using sensors they are replaced by the sensor inputs. Both the inputs and outputs of the graph are scaled to between `-1` and `+1`. This is to keep the values somewhat reasonable compared to the values the motors should take.

3.3.2 Evolutionary setup

In table 3.1 I will list the most important properties of my evolutionary setup. Most of the settings are fairly standard choices, others are mainly following the example of [12], described in section 2.3.1 on page 14. I will mainly use mutation, as it is preferred for CGP. For such long chromosomes and high probability GAlib checks for mutation on each element in the chromosome. Crossover is investigated in [23], implementing this would be a possible addition.

Table 3.1: Evolutionary setup

Population	100 individuals per generation.
Generations	100 generations limit.
Mutation rate	0.25 (probability).
Crossover	Not used.
CGP size	5×5 CGP nodes.
Fitness	Distance from starting point to ending point.

Chapter 4

Experiments

In this section I will describe each set of experiments, and discuss the results. Each subsection will have a Description section, a Results section and a Discussion section. The interesting points that are discovered during the experiments will be discussed in the Discussion and Conclusion, section 5 on page 50.

4.1 Presentation of the experiments

I have performed 8 distinct sets of experiments. The first 4 are the core experiments where I investigate different node function sets and other system changes. In the last 4 I dip into certain possible other directions to take this work. So the experiments from section 4.6 on page 43 and out are rather small sets. They are not as thorough as the core sets, and are only intended as a short look at things that might be interesting.

I have tried to mostly keep the description and results sections purely descriptive, keeping the discussion and evaluation of results for the discussions. For certain experiments it is more appropriate to let these flow a bit into one another.

The focus is on the behavior of the robots, and what noteworthy features I find in using CGP for the control system. I will be putting less weight on the specific internal workings of the system. In some cases I will to a small extent discuss the inner workings of the system, where it might help in understanding the behaviors of the robots.

4.1.1 The different sets of experiments

I have run several different sets of experiments, adding or changing different features of the system. In table 4.1 on page 29 I list the the different node function sets, and which ones are used in what experiment set.

- Main experiments sets:
  - Testing different node functions.

  - Different/new physics settings (friction, joint limits etc).
  - A sensor on the back marking cheat if they fall over and thus gets a better score.
  - Smaller CGP graph. Lower mutation rate. Both.

- Small exploratory experiment sets:
  - Two sine wave nodes with different phases (to see if improves emergence of gaits).
  - Stair obstacle set.
  - Random boxes obstacle set.
  - Joint angle sensors.

Each set was run with several different node function sets. Each run three times with different random seeds.

4.1.2 Measure of results

The best individuals of each evolutionary run is examined visually. First the behaviors of the robots are described, often first as a list of the different observed behaviors, then the most interesting ones are discussed in some detail. When visually examining the robots they are often run for longer then they were during evolution. Sometimes this shows behavior that would have been classified as cheating. In those cases they will be counted as having cheated, although that is not strictly the case. The highest fitness score for each separate run will be given in a table for each experiment set. The runs will be labeled with the node function and a number between 1 and 3 signifying the separate random seeds. For those robots that is shown with a picture, a video has also been uploaded to Youtube, in http://www.youtube.com/CGPQuadrobotMaster. A link to the specific video is provided in the caption of the relevant picture. The fitness values over generations will in some cases be plotted, in those cases where it might give some insight into how the solutions developed, and to compare runs to each other. For particularly interesting solutions diagram of the CGP will be shown, created with Sekaninas CGP Viewer [24]. The CGP diagrams are often difficult to interpret as they are not very human readable (and are not meant to be). A full mapping of what each of the CGP graphs perform in functions is somewhat outside of the scope of this thesis, so I will only make some short observations on the ones controlling the most interesting of robot behaviors.

4.2 Different node function sets

In this first set of experiments the system was tested with different node functions. First a rather basic set of functions was run as a comparison. Then tests with different function sets was run, usually with two or three additional node functions in addition to the basic ones. They are shown here in the order in which they were run.

Table 4.1: Experiment system and environment components

Node Functions	New physics	Anti-cheat	Lower mutation rate & Smaller CGP	Two Sine phases	Stair Obstacle	Random Box Obstacles	Joint Sensors
Basic	✓	✓	✓	✓	✓	✓	✓
Evolved Constant	✓	✓	✓	✓	–	–	–
Compare	✓	✓	✓	✓	✓	✓	✓
Sine and Cosine	✓	✓	✓	✓	✓	✓	✓
Logarithms	✓	✓	✓	✓	–	–	–

4.2.1 Description

There are five different function node sets in total, here numbered with the basic set as zero.

- The basic function set contained the functions: `+`, `-`, `*`, `/`, `wave`, `min` and `max`. All function nodes take two inputs and give one output, except the sine wave node, as it takes no inputs and gives one output. The sine wave node gives output between -1.00 and +1.00 float, and varies in a sine wave form according to the current simulation time, and the set values for the attack and decay of the wave. This is an almost direct conversion of the parameter optimization control system from [1, 18] output as a CGP node. This set is thought of as a base to compare the other different node function sets against. The functions seem to be the most common ones in use judging by other work ([12, 15]). The comparisons are plotted as dotted lines in the relevant fitness plots.

- In the first test with additional node functions I added a node with evolved constants, so that the function set was: `+`, `-`, `*`, `/`, `const`, `wave`, `min` and `max`. The const node takes no inputs and gives one evolved integer as output.

- The second I added absolute and compare nodes. The node function set is then `+`, `-`, `*`, `/`, `abs`, `compare`, `wave`, `min` and `max`. The absolute node takes a single input and gives one output, the compare node takes two outputs and gives `+1` if the first is greater, `-1` if the second is greater and `0` if they are equal.

- In the third `sin(a+b)` and `cos(a+b)` was added, which made the full set: `+`, `-`, `*`, `/`, `sin(a+b)`, `cos(a+b)`, `wave`, `min` and `max`. Both the sine and cosine nodes take two inputs and gives a single output.

- In the fourth set `exp`, `logN` and `log10` were added. This made the full set: `+`, `-`, `*`, `/`, `exp`, `logN`, `log10`, `wave`, `min` and `max`. All three of the new nodes (exponential, natural logarithm and base 10 logarithm) take one input and give one output.

4.2.2 Results

Each set of node functions was run three times with a different random seed each time. For the basic set all three runs produced robots that moved by twisting the central joint, and helping slightly with the other legs. For the evolved constant set of runs the first produced a similar gait as the basic runs, the last two made use of a friction inaccuracy to slide along the ground, shown in figure 4.1 on the facing page on the right. For the compare set of runs the first also made use of the friction cheat. The last two had a similar gait to the ones from the basic runs. For the sine/cosine runs the first two produced a similar movement as the basic runs, the last one jumped by sudden flexing of all four legs, shown on the left in figure 4.1 on the next page. For the logarithm runs all three produced the gait similar to the one from the basic runs.

Table 4.2: Fitness Values: Node Functions

Run	Max Fitness Old Scoring	Note
Basic01	1.1926	–
Basic02	1.4839	–
Basic03	1.1787	–
Const01	1.3022	–
Const02	2.2507	Cheat
Const03	2.2674	Cheat
Compare01	2.3029	Cheat
Compare02	1.4673	–
Compare03	1.2531	–
SineCos01	1.3729	–
SineCos02	1.1224	–
SineCos03	2.1421	Cheat
Logarithm01	1.1705	–
Logarithm02	1.1382	–
Logarithm03	1.1050	–
Stairs01	1.7639	Cheat
Stairs02	1.4839	–
Stairs03	1.4839	–

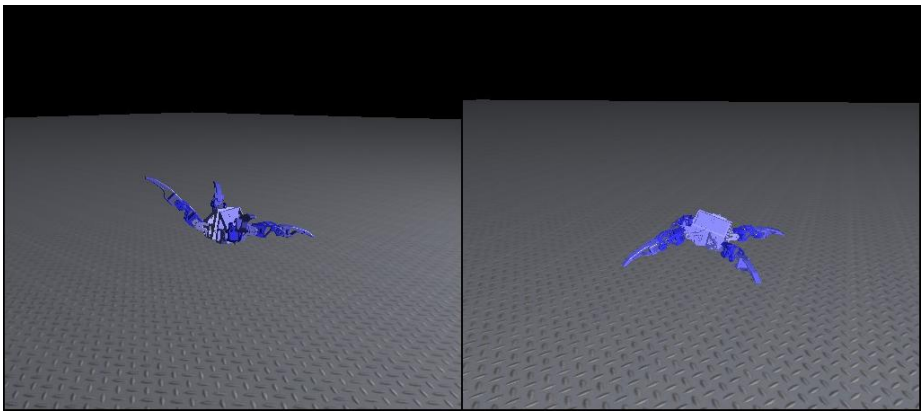

Figure 4.1: The Jumping robot on the left and Sliding robot on the right, modes of movement seen in the simulator. A video of the Jumping Robot can been seen at http://youtu.be/U3Y1mLTWfiY, and a video of the Sliding Robot can been seen at http://youtu.be/zR2JqOVp8dE.

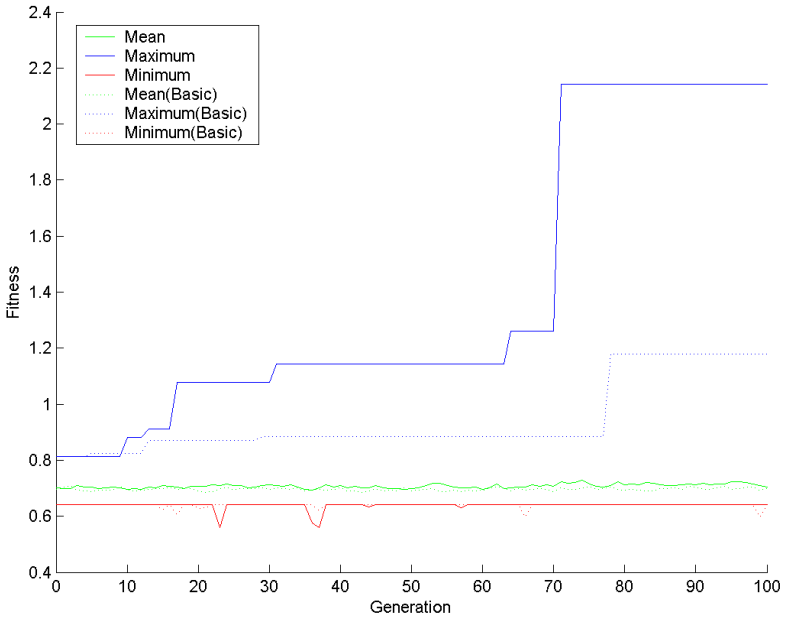

Figure 4.2: Fitness plot. Results of SinCosFunc03.

4.2.3 Discussion of results

I got three distinct modes of movements in this run:

- Walking by diagonal movement of the legs/central joint, causing slight movement, seems to be a rather ineffective gait.
- Jumping behavior. It had developed jumping by rather sudden flexing of all four legs.
- Physics cheat. Certain robots would set their legs at a particular angle to the ground plane, lean a bit to the side, and due to some inaccuracy in the friction calculations, they would slide along the ground, as shown in figure 4.1 on the preceding page on the right.

The most interesting modes of movement were that some of the robots developed jumping (shown in figure 4.1 on the previous page on the left). Such quick movements were unrealistic in the real robot, and this, in addition to the friction cheat made us run a new set of experiments with an updated set of physics variables and settings.

Jumping behavior was found in the third run with sine and cosine node functions (SinCosFunc03), with diagrams shown in figures 4.3 on the facing page and 4.2. It had developed jumping by rather sudden flexing of all four legs.

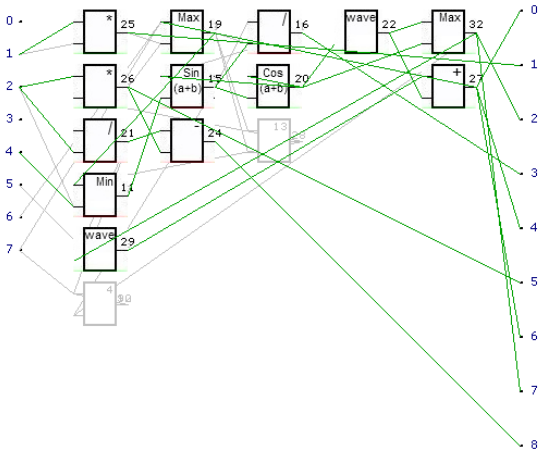

Figure 4.3: CGP diagram. Results of SinCosFunc03.

We can try to parse what the nodes do. We can see two wave signals being added together, before being sent to three of the legs. This might have made the signals curves much sharper.

The gaits consisting of almost only twisting the central joint might be a sign that this basic node function set is too limited to be able to produce more effective gaits, or a sign that the search space is too difficult (discussed further in section 5.1.2 on page 51), and that the evolution therefore should have been run for longer, over more generations, or with larger population.

4.3 New physics settings

Due to inconsistencies in the physics calculations many of the robots in the previous set of experiments were unfeasible. We needed better and more accurate variables for joint motor limits and speed as well as friction variables and other physics settings. The same various node function sets were all tested in this run as well.

4.3.1 Description

In the time since I received a copy of the system (see section 2.3.7 on page 18) there had been done separate development on the system for different work, including many improvements to the physics settings. Most of these could almost directly be transferred, except for the changes related to the outputs of the control system.

Most of the physics changes had to do with frictions and limitations on the motors. These were very promising changes to prevent the physics cheats encountered. The friction between the ground plane and the legs of the robot

were increased, and an anisotropic friction material was added to the legs. The densities of parts of the robot were fine tuned. The speed, maximum force and outputs to the joint motors were restricted. The fitness measure was changed to cm moved per second. And the evaluation time was increased to 7 seconds, up from 5 seconds.

Table 4.3: Fitness Values: New Physics

Run	Max Fitness	Note
BasicFunc01	7.13478	–
BasicFunc02	9.84492	Cheat
BasicFunc03	9.08	Cheat
ConstFunc01	11.849	Cheat
ConstFunc02	12.4568	–
ConstFunc03	10.2529	Cheat
CompareFunc01	10.2806	Cheat
CompareFunc02	11.3935	Cheat
CompareFunc03	8.72224	Cheat
SineCosFunc01	8.59215	Cheat
SineCosFunc02	8.03487	Cheat
SineCosFunc03	8.92931	Cheat
LogarithmFunc01	12.1208	Cheat
LogarithmFunc02	11.7126	Cheat
LogarithmFunc03	10.4809	Cheat
BasicFuncStairs01	8.2041	Cheat
BasicFuncStairs02	7.88693	Cheat
BasicFuncStairs03	8.69331	–

4.3.2 Results

By visually inspecting the robots I found that their behaviors could be divided into three groups.

Types of solutions:

- Cheating by falling.
- Cheating by getting as tall as possible (possibly related to the first point).
- Walking by pulling the body along with two front legs.

Most of the best individuals cheated in very similar ways. They would in slightly different ways lift the body up as high as possible, with the legs pointing almost directly down (shown in figure 4.4 on the facing page on the left), and then fall over (on the right in figure 4.4 on the next page). A small number of the robots only lifted the bodies high, and then stopped moving.

Three of the robots had somewhat promising gaits. This consisted of pulling with two front legs, and either leaving the other legs limp, or helping to some small extent. One example shown in figure 4.5 on page 36.

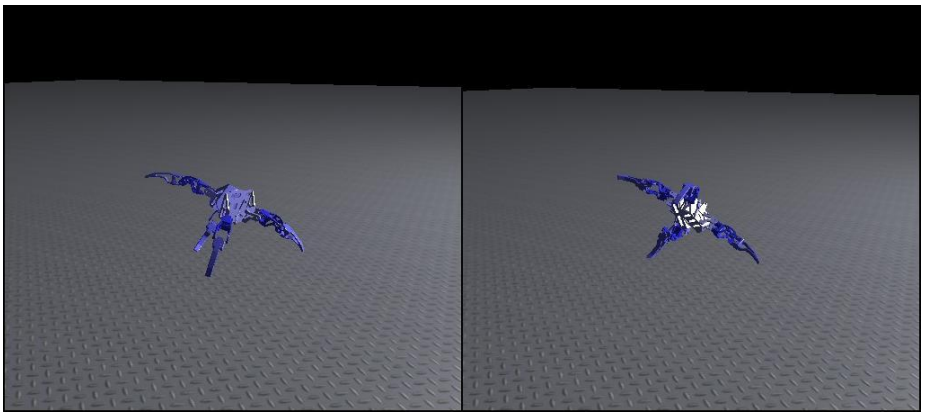

Figure 4.4: A robot lifting itself as high above the ground as possible (on the left) and then falling (on the right), from the simulator. A video can been seen at http://youtu.be/qxNJqgR6_H8.

4.3.3 Discussion of results

These changes removed the friction cheats and the jumping from the previous set, but instead two thirds of all best individuals cheated by lifting themselves as high as possible, and falling over on their backs, thus reaching further from the starting point faster (see figure 4.4 on the right). I would like to prevent them from falling, as many of the cheating solutions looked quite promising before falling over.

In figure 4.5 on the following page we have the best of the walking robots, BasicFunc01. It moves at rather good speed. One remarkable feature is that one of the legs have the lower part raised. This might just have happened to be that way, and never changed, or it might have avoided the further complexity of adding the rest of the leg to the problem. This solution gets higher fitness than many other solutions, see figure 4.6 on the next page. We can try to analyze the CGP diagram. In figure 4.7 on page 37, we can find two sine wave functions, but it is difficult to glean anything more from it, as this graph is quite convoluted. Many nodes, in particular the wave nodes, take their own outputs as input. The wave nodes do not use any inputs, but mutation might change the nodes function, this might lead to interesting results.

4.4 Penalizing cheating

In order to prevent the very prevalent cheating in the second run of experiments a type of cheat sensor could be used. On some of the other robot models in the system a single touch sensor (see section 3.1.2 on page 22) had been placed on the top (or back) of the robot and would, if triggered, set a cheat flag that would set the fitness score of the robot to zero. For this run of experiments such a sensor was placed on the back of the robot. The same various node function sets were used.

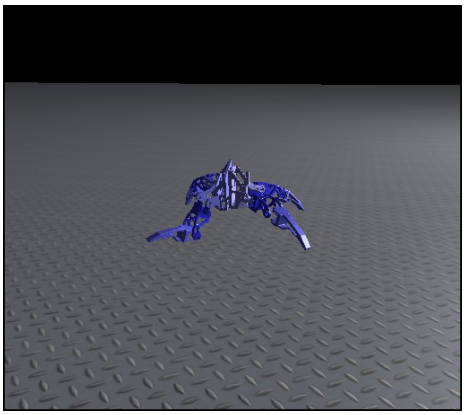

Figure 4.5: Picture. Results of BasicFunc01 Shows a somewhat effective gait, though one of the lower legs is lifted. A video can been seen at http://youtu.be/lpXQYqkyW9U.

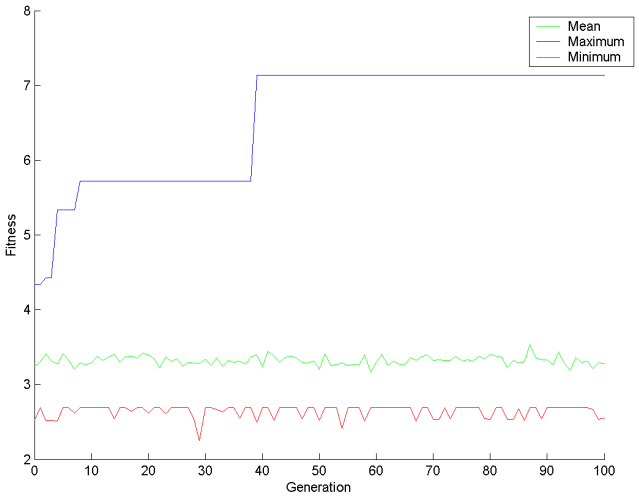

Figure 4.6: Fitness plot. Results of BasicFunc01.

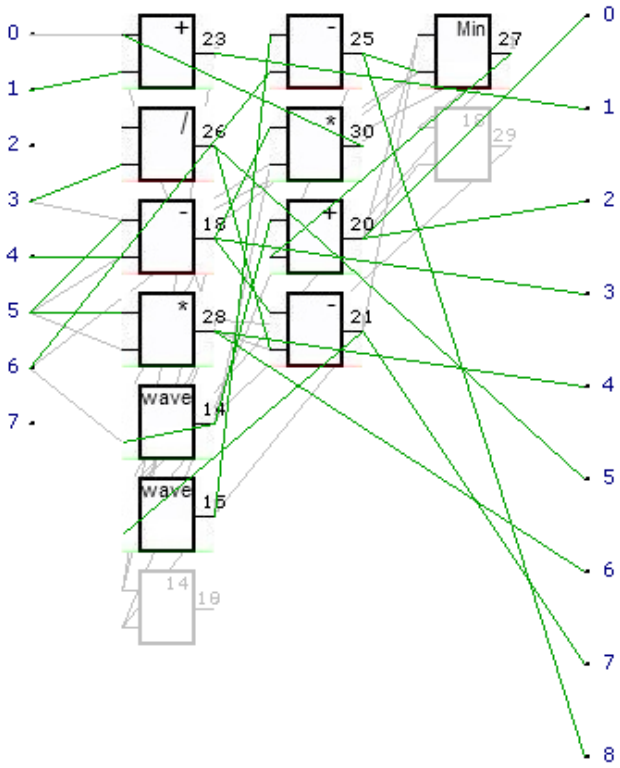

Figure 4.7: CGP Diagram. Results of BasicFunc01.

4.4.1 Description

The system is essentially the same as in the previous run. As so many of the solutions showed promise before falling over, it would be interesting to see what they would do if prevented from falling. The only change to the system is the addition of a sensor above the central joint giving the robot a fitness score of 0 if triggered. This sensor is separate from the sensor inputs to the CGP graph (see section 3.3.1 on page 25 for a description of this), this one is set by a PhysX Trigger Report, and only read in the fitness calculation.

Table 4.4: Fitness Values: Anti-Cheat

Run	Max Fitness	Note
BasicFunc01	7.05765	–
BasicFunc02	8.01878	–
BasicFunc03	6.52225	–
ConstFunc01	9.12928	Cheat
ConstFunc02	11.2018	Cheat
ConstFunc03	7.97803	Cheat
CompareFunc01	8.84398	Cheat
CompareFunc02	7.03025	–
CompareFunc03	7.5392	Cheat
SineCosFunc01	7.07456	–
SineCosFunc02	6.78356	–
SineCosFunc03	7.18914	Cheat
LogarithmFunc01	7.41777	–
LogarithmFunc02	7.37514	–
LogarithmFunc03	7.48069	Cheat
BasicFuncStairs01	5.95441	–
BasicFuncStairs02	7.08479	–
BasicFuncStairs03	7.61232	–

4.4.2 Results

In this set of experiments there were four distinct solutions:

- Rising up as high as possible (and eventually falling if run for longer than the evaluation).
- Falling but avoiding to trigger the sensor.
- Walking by pulling the rest along using one leg.
- Walking by pulling the rest along using two legs.

Just about all the different node function sets produced solutions of the first type. This is perhaps because the fitness measures distance in all directions from the starting point, so merely getting some small distance above it would give a slight increase in fitness. The evaluation is only run for 7 seconds, but when visually inspecting the robots, I run each of them manually, and can let

them run for however long. They move a small distance before falling, so it seems that the evolution selected not so much for moving far, but for length of time before falling over. It is to be expected that there would be some such solutions.

Basic, Compare and SinCos produced the ones with any sort of walking gait. In particular SinCos02 and to some extent SinCos01 showed gaits moving at very good speed. The gaits have some symmetry in that the movement alternates, though somewhat unevenly, between two front legs.

The logarithm set also produced some walking gaits, but in this the entire robot vibrated as well so they would not be feasible in the real robot.

One humorous solution fell over on the back and avoided to trigger the cheat sensor. This sort of solution was perhaps to be expected. Evolution will find an opening if one exists.

4.4.3 Discussion of results

A handful of the solutions from this set of experiments circumvented our constraints by either falling but not triggering the sensor, of by taking long enough to fall that the evaluation finishes. This is to be expected, and often we get many interesting, or at least entertaining solutions as a result of this. Constraints such as these also create a form of selection pressure, as we can see with the robots that instead of being selected for how far they move, instead were selected based on how long they could remain standing before falling.

We have some gaits that are both possible in the real robot, and that move at fairly good speed. We are however noticing more and more that we have rather little symmetry in the movement of the robot. CGP should allow or even encourage it, but it appears very seldom in the evolved robots. This will be further discussed in section 5.1.1 on page 50. The good gaits are walking with two 'front' legs. The movement is however very uneven. We still lack a form of 'true' symmetry, but it is promising that some symmetry appears spontaneously. The next set of experiments is then an attempt to make the search space easier, smaller and more even (this is discussed further in section 5.1.2 on page 51), with the purpose of investigating whether that will give better and possibly more symmetrical solutions.

4.5 Smaller CGP and lower mutation rate

It seemed that the robots very rarely got to very good solutions. Some might be because of a difficult search space and some because of the rather limited joints of the robot (see sections 5.1.2 on page 51 and section 3.1.1 on page 22 for further discussions of these problems). In order to make the search space less complex and smaller one test with lower mutation rate was run, one with a smaller CGP in number of nodes, and one run with both changes. Running with lower mutation rate would also make the evolution jump less between, or

away from, local maxima. This property of CGPs is discussed in section 3.3.1 on page 25.

4.5.1 Description

Running the same node functions as previously. The system is mostly the same as earlier, the only changes are that the mutation rate is decreased from 0.25 to 0.10, and the size of the CGP is lowered from 5*5 nodes to 4*3 nodes.

Table 4.5: Fitness Values: Small CGP

Run	Max Fitness	Note
BasicFunc01	8.52473	–
BasicFunc02	9.6673	–
BasicFunc03	10.3079	–
ConstFunc01	10.3826	Cheat
ConstFunc02	8.67413	Cheat
ConstFunc03	8.95783	Cheat
CompareFunc01	8.49006	Cheat
CompareFunc02	9.50749	–
CompareFunc03	8.68544	Cheat
SineCosFunc01	6.97324	–
SineCosFunc02	10.4939	–
SineCosFunc03	8.40844	–
LogarithmFunc01	8.63286	–
LogarithmFunc02	7.63205	–
LogarithmFunc03	7.31128	–
BasicFuncStairs01	7.74124	–
BasicFuncStairs02	7.13478	–
BasicFuncStairs03	8.10202	–

4.5.2 Results

We can group the results into the following:

- Pulling itself forward using two legs.
- Using all four legs to walk, but very unevenly.
- Falling over after some form of ungainly gait.
- Falling over and then laying still.

We still get some solutions that fall over, and even some that don't move much, the only movement is to get the legs to some set position, that will make the robot fall.

One notable solution from the LowMut run avoids falling over by wrapping one leg over the body, and twisting the central joint, and then somehow managing to move forwards. This solution also finds a way around the limitation

Table 4.6: Fitness Values: Lower Mutation Rate

Run	Max Fitness	Note
ConstFunc02	10.3002	–
ConstFunc03	9.27638	Cheat
CompareFunc01	8.62102	–
CompareFunc02	8.66237	–
CompareFunc03	7.13478	–
SineCosFunc01	9.70589	Cheat
SineCosFunc02	7.10112	–
SineCosFunc03	6.26539	–
LogarithmFunc01	7.48069	Cheat
LogarithmFunc02	10.9833	–
LogarithmFunc03	7.42258	–
BasicFuncStairs01	6.52656	–

Table 4.7: Fitness Values: Both Smaller CGP and Lower Mutation Rate

Run	Max Fitness	Note
BasicFunc01	9.6673	Cheat
BasicFunc02	8.80113	–
BasicFunc03	8.84556	–
ConstFunc01	9.55013	Cheat
ConstFunc02	8.71384	–
ConstFunc03	9.55013	Cheat
CompareFunc01	8.82323	–
CompareFunc02	8.93451	Cheat
CompareFunc03	12.0168	Cheat
SineCosFunc01	8.01593	Cheat
SineCosFunc02	8.48741	–
SineCosFunc03	8.18076	–
LogarithmFunc01	7.72417	–
LogarithmFunc02	9.64805	Cheat
LogarithmFunc03	8.67497	Cheat
BasicFuncStairs01	7.19564	–
BasicFuncStairs02	8.84854	–
BasicFuncStairs03	7.86811	–

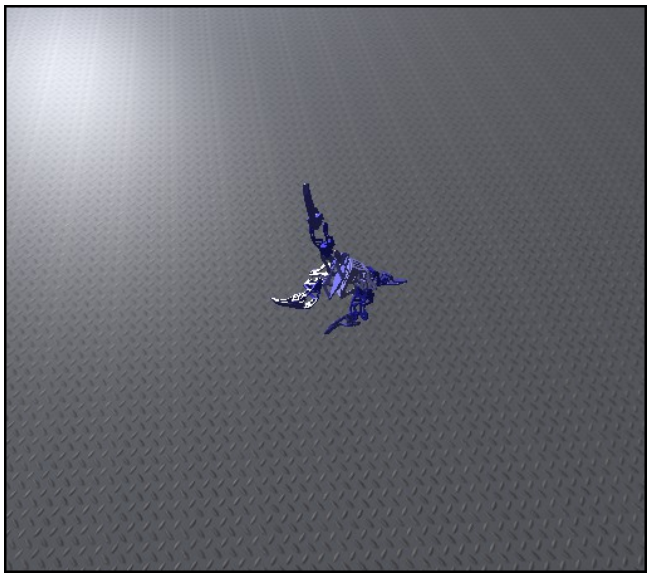

Figure 4.8: Picture of robot avoiding falling by wrapping the legs around the body. Results of Logarithm03. A video can been seen at http://youtu.be/24vbCmDsaak.

on the movement of the legs, as seen in the picture 4.8, one of the legs can now move horizontally due to the permanent twist of the central joint. It is quite humorous to find such solutions circumventing your rules in interesting ways. And maybe there is something to be learned from it as well. When these robots fall onto their backs they don't have any ways to get back up again. So perhaps something like this would be a useful addition to the robot shape (or morphology), perhaps a pyramid shape on top of the robot, to prevent it from falling over too far, so that it might recover from a fall, and continue its function.

From the small CGP set many of them would fall over on the side (thus avoiding triggering the cheat sensor) and then lay still. But from this same run a surprisingly large group of similar well functioning walking gaits appeared. There seemed to be very little variation in the solutions however. As discussed in 3.3.1 on page 25 the constant jumping into and away from local maxima is often a desirable feature of CGPs, though sometimes it is also a problem.

4.5.3 Discussion of results

One rather noticeable feature was revealed in this run: Smaller CGP leads to more similar results (less variation). This can be both good and bad. One good effect of this is that we seem to get more symmetry in the smaller CGP runs, presumably because of greater reuse of nodes.

There did not seem to be any significant improvement when running both changes, though each set did have improvements, more tuning will be needed

for there not to be too many disadvantages from the changes. I will discuss this further in section 5.1.2 on page 51.

4.6 Two sine wave nodes

Though symmetry is possible in the system we don't explicitly encourage it, in the hope that evolution itself would reach such solutions. This however often does not seem to be the case. There could be many reasons for this (see section 5.1.1 on page 50 for further discussion) but one possible aid would be to instead of having a single sine wave node, we would have two, going at different phases.

4.6.1 Description

Same system as the AntiCheat run 4.4 on page 35. To each of the node function sets a second sine wave node was added. No further changes were made. The second sine wave node uses all the same settings as the first, but the phase is half a cycle earlier.

Table 4.8: Fitness Values: Two Sine Nodes

Run	Max Fitness	Note
BasicFunc01	8.11701	–
BasicFunc02	6.40523	–
BasicFunc03	8.11701	–
ConstFunc01	9.76787	Cheat
ConstFunc02	10.0231	Cheat
ConstFunc03	10.0231	Cheat
CompareFunc01	9.37251	–
CompareFunc02	8.11701	–
CompareFunc03	8.83788	Cheat
SineCosFunc01	6.20021	–
SineCosFunc02	6.58036	–
SineCosFunc03	7.06023	Cheat
LogarithmFunc01	8.11701	–
LogarithmFunc02	7.31682	–
LogarithmFunc03	8.83788	Cheat
BasicFuncStairs01	6.25832	–
BasicFuncStairs02	8.67345	–
BasicFuncStairs03	8.5919	–

4.6.2 Results

Groups of results:

- Getting as tall as possible, then falling.
- Pulling itself forward by one leg.

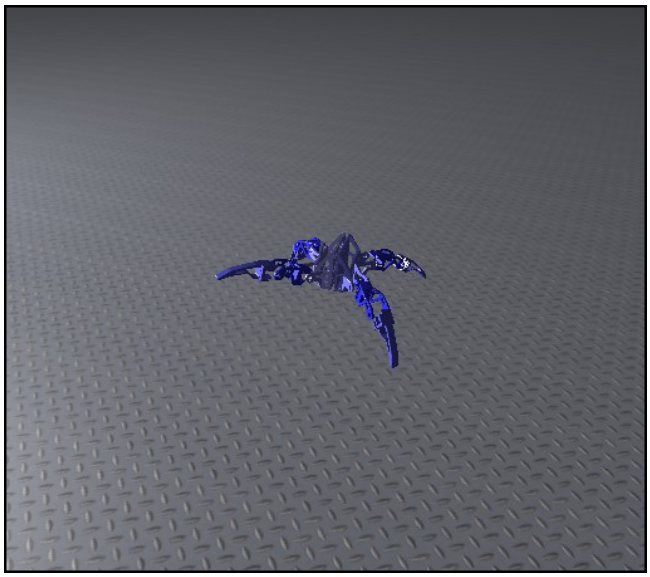

Figure 4.9: Picture of robot walking in a manner somewhat similar to a turtle. Results of SineCosFunc02. A video can been seen at http://youtu.be/13-ZiHC4C0Q.

- Pulling itself forward by two legs.
- Using all four legs.

Many of the solutions move by pulling the body forwards using two legs. However the majority of the solutions mainly use a single leg for movement, with the other legs helping things along.

One notable one 4.9 moves like a turtle, and uses all four legs for movement.

4.6.3 Discussion of results

Slight improvement over having a single node, but not as great improvement as having a smaller CGP proved to be. Having two sine nodes seemed to make the search slightly easier. But the is still unevenness in the motion of the robot. One could call this more of a band-aid, and less of a solution. There is in this set as well a portion of the solutions that cheat by falling. I have chosen not to pursue more ways of limiting this behavior as I deemed it to be rather little gain from the time spent.

4.7 Stair obstacles

Along with the usual experiments on a flat ground plane I also ran some of the setups with stairs.

4.7.1 Description

We have stairs surrounding the robot starting point at 100 cm distance, with 40 cm between the steps. This setup of stairs is somewhat inspired by Miikkulainens fences in [21]. The fitness measure is still distance moved from start, so stairs like these, surrounding the robot, will have a sort of two stage fitness effect, in that first the robot will be rewarded for moving away from the start, then as it encounters the stairs it cannot get higher fitness unless it climbs them.

These tests with stairs is not a separate set of runs as the other experiments have been. These have been run as a small part of almost all the other sets, for the sake of variety. So some of the results are form the earliest function node runs, others are from the runs with smaller CGP and so on.

Table 4.9: Fitness Values: Stairs (repeated from the other tables)

Run	Max Fitness Old Scoring	Note
Node Functions:		
`Stairs01`	1.7639	Cheat
`Stairs02`	1.4839	–
`Stairs03`	1.4839	–
New Physics:		
`BasicFuncStairs01`	8.2041	Cheat
`BasicFuncStairs02`	7.88693	Cheat
`BasicFuncStairs03`	8.69331	–
Anti-Cheat:		
`BasicFuncStairs01`	5.95441	–
`BasicFuncStairs02`	7.08479	–
`BasicFuncStairs03`	7.61232	–
Small CGP:		
`BasicFuncStairs01`	7.74124	–
`BasicFuncStairs02`	7.13478	–
`BasicFuncStairs03`	8.10202	–
Lower Mutation Rate:		
`BasicFuncStairs01`	6.52656	–
Both Smaller CGP and Lower Mutation Rate:		
`BasicFuncStairs01`	7.19564	–
`BasicFuncStairs02`	8.84854	–
`BasicFuncStairs03`	7.86811	–
Two Sine Nodes:		
`BasicFuncStairs01`	6.25832	–
`BasicFuncStairs02`	8.67345	–
`BasicFuncStairs03`	8.5919	–

4.7.2 Results

In short the types of solutions are:

- Jumping.
- Falling.
- Walking in circles.
- Gets stuck in a corner.
- Falling when colliding with the first step.
- Almost getting up the first step.
- Getting up the first step, but gets stuck there.

It seems that regardless of which run of experiments the results are from we get the same types of results. One exception is the jumping, as it could only occur before the new physics settings were added. A small handful of solutions do no climbing whatsoever and only cheat by falling. Many walked decently but got stuck in the corner of the stairs, never getting up on a step. The rest got up one step but no more.

4.7.3 Discussion of results

I ran the stair obstacles alongside many of the other sets of experiments, for the sake of variety. Most of the solutions evolved never even reached the stairs, let alone climbed them. This could perhaps be expected, as I did not do any adjustments to the setup before running it with stairs, the fitness evaluation remained the same, as well as all other components of the system. But this was also the intention, to see if the robots evolved in the other runs could be used directly in stairs. And some worked surprisingly well.

However, even the best of the stair climbing solutions only got as far as the first step. The stairs currently used are probably too steep, and if I would delve properly into this angle of study I would like to make several sets of stairs, both less seep and steeper, and with higher and lower step height.

4.8 Random Box obstacles

In this section I will describe the results with random box obstacles. The purpose of this set is to see how well certain of the gaits/function sets behave if there is an obstacle in the way. Will it simply fall over of the back, or will it handle the change of direction well, and simply continue in the other direction?

This set of experiments is run with a smaller collection of function node sets. The Compare and the SineCosine sets showed the most promise in earlier tests, so these, along with the Basic set, for comparison, were the only function sets used in these tests.

4.8.1 Description

The obstacles used here are randomly placed boxes of sizes 10 to 30 cm breadth and depth, and half that in height. The boxes were permitted to intersect with each other. The purpose of this was to create some somewhat concave shapes, to better investigate how the system would deal with such more complex obstacles. Walls were placed at 200 cm distance from the center. An area in the middle of the room was left free of obstacles, so that the robot would not land in or get stuck on a box before the evaluation started.

Table 4.10: Fitness Values: Random Obstacles

Run	Max Fitness	Note
BasicFuncRand01	6.56303	–
BasicFuncRand02	6.76494	–
BasicFuncRand03	6.43998	–
CompareFuncRand01	7.03025	–
CompareFuncRand02	5.85933	–
CompareFuncRand03	7.40206	Cheat
SineCosFuncRand01	10.7641	Cheat
SineCosFuncRand02	8.50615	Cheat
SineCosFuncRand03	6.55535	–

4.8.2 Results

For this run of experiments I got these five groups of solutions:

- No gait, only setting the legs to a position where it falls over (without triggering the sensor).
- Walking a small distance and falling over.
- Walking and falling when colliding with a wall.
- Pulling the body forwards using one leg, the other legs helping, getting stuck in a concave corner.
- Walking using two legs and getting stuck at a convex corner.

We still get some solutions that move farthest by falling over. It seems that this is difficult to avoid, and would probably require a more limiting fitness function. The ones falling when colliding with a wall came as no surprise, this is the sort of thing I wanted to investigate with this set of tests. Interestingly however I got some solutions that get stuck on concave corners and some that get stuck on convex ones.

4.8.3 Discussion of results

With this small number of runs I cannot say of certain, but it seems that those that move using primarily one leg gets stuck on concave corners and those using two gets stuck on convex ones. It would make sense considering the shape of the

robot. The single leg pulling would pull it into a corner, while two legs would get stuck on either side of one.

4.9 Joint angle sensors

I had long intended this system to be used along with sensors, but as the focus of the thesis shifted from finding interesting behaviors by using sensors towards more on evaluating the suitability of CGP for such a system, many of these ideas were abandoned. However one small run was performed, using joint angle sensors.

4.9.1 Description

The same system as with the new physics run. No obstacles were used. Instead of the normal 0.5 on all inputs to the CGP (see section 3.3.1 on page 25 for explanation) we read the joint angles from PhysX. The purpose of this test is to investigate if there is some improvement, such as some useful effect of the feedback between the outputs and inputs of the CGP.

Table 4.11: Fitness Values: Joint Angle Sensors

Run	Max Fitness	Note
`BasicFuncJoint01`	7.79597	Cheat
`BasicFuncJoint02`	7.37688	Cheat
`BasicFuncJoint03`	6.95835	Cheat
`CompareFuncJoint01`	7.19301	–
`CompareFuncJoint02`	8.389	–
`CompareFuncJoint03`	7.79078	Cheat
`SineCosFuncJoint01`	7.16488	Cheat
`SineCosFuncJoint02`	8.53707	–
`SineCosFuncJoint03`	9.49001	–

4.9.2 Results

These are the different types of solutions I got from this run:

- No gait, static falling over.
- Rising up tall, falling over.
- Moving in circles using one leg.
- Using all four legs, rather slow progress, moving in a circle.
- Quite good speed of movement using one leg and the central joint.
- Cheating by spinning (physics miscalculation).

We still get many solutions falling over by various means. We have some that move in a circle, by mainly pulling the body along using one leg, one with all the legs helping it along. It is still a rather slow mode of movement however. We have one quite fast gait, SineCos02. It moves by pulling the body along using one folded leg, and twisting the central joint so that the other legs walk in a sort of symmetrical gait. We also got one new type of cheat. The robot would set the legs in a particular position, and as it touched the ground plane the robot would spin rather fast around and away from the center, to eventually fall to rest. It is difficult to diagnose what might be the cause of this, but presumably it is a particular configuration that causes the anisotropic friction material calculations to behave in this manner.

4.9.3 Discussion of results

All in all not many noticeable differences from the earlier setups. This line of work would need more investigation and adjustments to find out if there is any useful effects of this type of feedback.

Chapter 5

Discussion

In this chapter I will first discuss the most noticeable properties of my results, thin I will mention some possibilities for future work, and finally I will round off with the conclusion.

5.1 Concluding discussion

I decided to implement CGP because I wanted to have a system that was open and that did not impose any assumptions or limitations on what shape the solutions might take. A necessary side effect of this is that the search space becomes much more complex than with a system that does make assumptions. A system that imposes limitations on the solutions will have a smaller and more limited search space, but might also never get to the interesting or innovative solutions, as the designer limits it to his own knowledge.

5.1.1 Lack of symmetry

All the solutions had irregularities in the gaits produced, and even the best of the gaits looks like they would have been improved by having these irregularities removed. But they might be a necessary side effect of evolving the gait in such an open way as I do. I evolve the movement of each joint mostly separately, if two joints use the same calculations then that is by chance, and that is how I designed the system.

I however did get some examples of good symmetry, in particular in those with a smaller CGP graph. Further work would perhaps have found an optimal size of graph.

The shape of the robot (see section 3.1.1 on page 22 on the morphology of the robot) is quite limited. The joints limit to a very large degree the types of gaits we can achieve. The morphology of the robot brings with it a great deal of assumptions on what shape the solution can take. So a control system that is also limited, and that makes perhaps the same sort of assumptions is more appropriate to this robot. The parameter optimization control system used in [18, 20] is an example of this. My system is much broader than that, and should

make few assumptions. It is my hope that this system will be used on a variety of different robot morphologies, and that more improvements can be made to it. It is very simple to fit this control system to different robots, the only things that need to be changed is the number of inputs and the number of outputs to the CGP graph.

Miikkulainen in [21] mentions that in systems that do not explicitly force symmetry the gaits produced resemble crippled animals. He examines the usefulness of pairing the legs (of a four-legged robot) in groups seen in real animals, like trot (diagonal legs), pace (same legs) and so forth. In my system symmetry did emerge, but with irregularities that sometimes prevented the solution from functioning as well as the same gait perhaps would have if we forced true symmetry. In real animals symmetry is encouraged as legs easily fall into groups, and are moved synchronously.

5.1.2 Difficult search space

It seems that the search space is large, difficult and uneven for most for the runs. We can see one sign of the unevenness in the plateaus in the fitness plots. Having a smaller CGP makes the search space much smaller, though it would still be very uneven. Having even fewer possible node functions would also make the search space simpler. Possibly a major cause is not restricting the connectivity of the CGP graph. As previously mentioned with such a limited robot, a more limited system would be more appropriate. It is likely that another of the main causes of the search space is the limitations on the robot itself. So having a more open, or even evolved robot morphology would be more appropriate for this system.

5.1.3 Advantages and disadvantages of CGP

All in all I have seen very clearly many of the advantages and disadvantages of CGP. The search space did get difficult, the modularity and reuse of nodes had its own uses and problems. The possibility of a single mutation to change the functioning of the graph completely has both been a very clear advantage and disadvantage. It kept the search from getting stuck in local maxima, but also disrupted the search. But it all served the purpose I had for it well, though much more fine tuning would be required to get things to go as smoothly as I would have liked.

I have very clearly seen both the beneficial and the detrimental effects of escaping local maxima in that the search might jump around so much that it never finds any good solutions.

5.2 Future Work

Throughout the work I found several points that would be promising for further study. The system could be expanded in a variety of ways. Here I will mention some of the ones I touched upon during my work.

5.2.1 Rewarding Symmetry

As a possible expansion on the system some manner of rewarding symmetry would perhaps improve the results further. Forcing symmetry goes against the purpose of a general system, but symmetry should perhaps be encouraged. Possibly also some manner of smoothing a gait once a good one has been found, to further improve it by removing irregularities.

5.2.2 Evolving morphology

Both in Sims work and in the Golem Project the morphology of the robots were evolved along with the control system. This would be an interesting expansion to this system, and I had plans for having this as a part of this work, but decided against it as the scope of the thesis would have become too wide. It would be more appropriate to have a more dynamic morphology for a robot with this system.

5.2.3 Evolving sensor morphology

Parker et al. evolved the positions of their sensors, as well as other properties of the sensors. One possibility for this robot is to let the possible positions be on each of the building blocks in the physics simulation. This would be most appropriate for sensors like touch sensors or light sensors. For distance sensors and tactile sensors that take the form of feelers having a platform on top of the robot is perhaps more appropriate.

5.2.4 Reproducing robots from simulation to reality

The Golem Project reproduced their robots from simulation to reality using a 3D printer. As more and more features are added to this system by others, adding this possibility would be one of the goals. If that were to be done the problem of the 'reality gap' would have to tackled. One possible change to this system in order to reduce the reality gap would be to have the obstacles slightly change position each run, to introduce noise and make the system more robust (inspired by Parker [7]).

5.3 Conclusion

In this work I have explored and seen the effects of using Cartesian Genetic Programming to control a simulated quadruped robot. The control system is tested and explored using a variety of node functions, a larger and smaller number of function nodes, changes in physics settings, and a handful of different obstacles and sensors. Some interesting solutions were found, such as successfully climbing stairs (though only up the first step in my experiments, but having the steps slightly further apart would likely improve this result). I also found a handful of very promising walking gaits, also some that are similar to earlier results using the same robot but with a different control system. However, I found that using such a general system might not the most appropriate for this rather limiting robot design. The added complexity gained from using this system made the search difficult, and reduced the chances of getting interesting designs. While the experiment setup would need further fine tuning I can recommend this system for use as a robot controller, though it would be more suited for a more general robot setup, or even an evolved robot morphology.

Bibliography

[1] Kyrre Glette and Mats Hovin. The x2 modular evolutionary robotics platform. In *Proceedings of the 9th International Conference on Evolvable Systems: from biology to hardware*, ICES'10, pages 274–285. Springer-Verlag, 2010.

[2] Karl Sims. Evolving virtual creatures. In *SIGGRAPH '94 Proceedings, Computer Graphics*, Annual Conference Series, pages 15–22, July 1994.

[3] Karl Sims. Evolving 3d morphology and behavior by competition. In Brooks and Maes, editors, *Artificial Life IV Proceedings*, pages 28–39. MIT Press, 1994.

[4] Nicolas Lassabe, Hervé Luga, and Yves Duthen. A new step for artificial creatures, 2007.

[5] Hod Lipson and Jordan B. Pollack. The golem project. In NATURE volume 406 pages 974-978, 2000.
http://demo.cs.brandeis.edu/golem/ (2010-09-29).

[6] L. France, A. Girault, J-D. Gascuel, and B. Espiau. Sensor modeling for a walking robot simulation. Eurographics99, 1999.

[7] Gary B. Parker and Pramod J. Nathan. Co-evolution of sensor morphology and control on a simulated legged robot. In *Computational Intelligence in Robotics 2007 (CIRA'07)*, pages 516–521, 2007.

[8] Gary Parker and Pramod J. Nathan. Response to changes in key stimuli through the co-evolution of sensor morphology and control. In *Proceedings of the 2008 World Automation Congress 7th International Symposium on Intelligent Automation and Control (ISIAC 2008)*, pages 1–8, 2008.

[9] Gary B. Parker and Pramod J. Nathan. Concurrently evolving sensor morphology and control for a hexapod robot. In *IEEE Congress on Evolutionary Computation'10 (CEC2010)*, pages 1–6, 2010.

[10] Karthik Balakrishnan and Vasant Honavar. On sensor evolution in robotics. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 455–460. MIT Press, 1996.

[11] Julian F. Miller and Peter Thomson. Cartesian genetic programming. In *Proceedings of the Third European Conference on Genetic Programming (EuroGP2000)*, volume 1802 of *Lecture Notes in Computer Science (LNCS)*, pages 121–132. Springer-Verlag, 2000.

[12] Simon Harding and Julian F. Miller. Evolution of robot controller using cartesian genetic programming. *EuroGP 2005*, LNCS 3447:62–73, 2005.

[13] Andrew L. Nelson, Gregory J. Barlow, and Lefteris Doitsidis. Fitness functions in evolutionary robotics: A survey and analysis. *Robotics and Autonomous Systems*, 57(4):345–370, April 2009.

[14] Corné Sprong. Common tasks in evolutionary robotics, an overview, 2011.

[15] Riccardo Poli, William B Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008.

[16] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 2007.

[17] Julian F. Miller, editor. *Cartesian Genetic Programming*. Natural Computing Series. Springer, 2011. DOI: 10.1007/978-3-642-17310-3 ISBN: 978-3-642-17309-7 http://www.springerlink.com/content/978-3-642-17309-7/contents/ (2011-10-03).

[18] Kyrre Glette, Gordon Klaus, Juan Cristobal Zagal, and Jim Torresen. Evolution of locomotion in a simulated quadruped robot and transferral to reality. In *Proceedings of International Symposium on Artificial Life and Robotics, AROB 17th*. Alife Robotics, 2012.

[19] Jason Yosinski, Haocheng Shen, Sean Lee, and Eric Gold. Quadratot. http://quadratot.yosinski.com/.

[20] Jason Yosinski, Jeff Clune, Diana Hidalgo, Sarah Nguyen, Juan Cristobal Zagal, and Hod Lipson. Evolving robot gaits in hardware: the hyperneat generative encoding vs. parameter optimization. In *Proceedings of the 20th European Conference on Artificial Life*, pages 890–897, 2011.

[21] Vinod K. Valsalam and Risto Miikkulainen. Modular neuroevolution for multilegged locomotion. In *Proceedings of the Genetic and Evolutionary Computation Conference GECCO 2008*, pages 265–272, 2008.

[22] Julian Miller. Cgp in c: Cartesian genetic programming implementation (library). https://sites.google.com/site/julianfrancismiller/cgp-software/CGP-version1_1.7z?attredirects=0.

[23] Janet Clegg, James Alfred Walker, and Julian Francis Miller. A new crossover technique for cartesian genetic programming. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1580–1587. ACM Press, 2007.

[24] Lukas Sekanina. Cgp viewer. http://www.fit.vutbr.cz/~vasicek/cgp/.

Program Code

This is a listing of the CGP code as well as other notable functions, such as the fitness scoring.

Listing 1: Code/CGP.h

```
1  #ifndef CGP_H
2  #define CGP_H
3  //inspired by Millers CGP implementation:
4  //https://sites.google.com/site/julianfrancismiller/cgp-software/CGP
       -version1_1.7z?attredirects=0.
5
6  #include "../cgp/CGPglobals.h"
7  #include "../cgp/CGPnodefunctions.h"
8
9
10 int* getUsedNodes(int* chromosome);
11 void printChromosome(int* printarray);
12
13 void decode(int* chromosome, double* inputData, double* graphOutputs
       )
14 {
15 //går fra inputs til outputs
16 //får inn kromosomet i en array.
17 //har en nodeOutputs array (diger) og en nodeInput array (på 3 eller
        største arity).
18   double nodeInput[MAX_ARITY] = {0.0};
19   double nodeOutputs[MAX_NUM_OUTPUTS] = {0.0};
20   int nodeBegin = 0;
21   int nodeBeginGene = 0;
22   int outIndex = 0;
23   int numNodesUsed = 0;
24   int functionIdentifier = 0;
25
26   int* usedNodes = getUsedNodes(chromosome);
27
28 //putt inputs(til grafen) øverst i nodeOutputs array
29   for (int i = 0; i < NUM_SENSOR_INPUTS; i++){
30     nodeOutputs[i] = inputData[i];
31     outIndex++;
32   }
33   for(int i=0; i<NUM_NODES; i++){
34     if(usedNodes[i] != 0)
35     numNodesUsed++;
36   }
37
38   for (int i = 0; i < numNodesUsed; i++)
39   {
40     if( i % NODE_LENGTH == 0) {
41       nodeBegin = usedNodes[i] - NUM_SENSOR_INPUTS;
```

```
42        for (int j = 0; j < NODE_LENGTH - 1; j++)
43        {
44          nodeInput[j] = nodeOutputs[chromosome[nodeBegin + j]];
45        }
46      }
47      functionIdentifier = chromosome[nodeBegin + NODE_LENGTH - 1];
48      nodeOutputs[nodeBegin + NUM_SENSOR_INPUTS] = nodeFunction(
            nodeInput, functionIdentifier, usedNodes[i] -
            NUM_SENSOR_INPUTS);
49    }
50    for (int i = 0; i < NUM_GRAPH_OUTPUTS; i++){
51      graphOutputs[i] = nodeOutputs[chromosome[CHROMOSOME_LENGTH -
            NUM_GRAPH_OUTPUTS + i]];
52    }
53    delete[] usedNodes;
54  }
55
56
57  int* getUsedNodes(int* chromosome)
58  {
59    const int nodeFlagSize = NUM_NODES + NUM_SENSOR_INPUTS + 1;
60    int nodeFlags[nodeFlagSize] = {0};
61
62    int *usedNodes = new int[NUM_NODES];
63    for ( int i = 0; i < NUM_NODES; i++ ){
64      usedNodes[i] = 0;
65    }
66  //går fra outputs til inputs, for å se hvilke noder som er i bruk.
67  //henter inputs fra hver node bakover
68  //merker med true i nodeFlags om noden er i bruk
69  //denne blir brukt til å merke posisjonene i arrayen som sendes
        tilbake.
70
71  //først setter nodene brukt av outputs til true.
72    for (int i = CHROMOSOME_LENGTH - NUM_GRAPH_OUTPUTS; i <
          CHROMOSOME_LENGTH; i++){
73      nodeFlags[chromosome[i]]=1;
74    }
75  //går bakover gjennom arrayen, om vi finner en true merker vi av
        dens inputs som true
76    for (int i = nodeFlagSize - 2; i >= NUM_SENSOR_INPUTS; i--){
77      if (nodeFlags[i]){
78        int index = NODE_LENGTH * (i - NUM_SENSOR_INPUTS);
79        for (int j = 0; j < NODE_LENGTH - 1; j++){
80          nodeFlags[chromosome[index + j]] = 1;
81        }
82      }
83    }
84  //Kjører igjen for å få med de som ikke ble med i stad.
85    for (int i = nodeFlagSize - 2; i >= NUM_SENSOR_INPUTS; i--){
86      if (nodeFlags[i]){
87        int index = NODE_LENGTH * (i - NUM_SENSOR_INPUTS);
88        for (int j = 0; j < NODE_LENGTH - 1; j++){
89          nodeFlags[chromosome[index + j]] = 1;
90        }
91      }
92    }
93  //når vi har vært gjennom hele, putter vi adressene(posisjonene) i
        ut arrayen.
94    int numberOfNodesUsed = 0;
95    for(int i = NUM_SENSOR_INPUTS + 1; i <= nodeFlagSize - 1; i++){
96      if (nodeFlags[i]){
```

```
97        usedNodes[numberOfNodesUsed] = i;
98        numberOfNodesUsed++;
99      }
100   }
101   return usedNodes;
102 }
103
104
105 GARealAlleleSetArray setupAlleles()
106 {
107   GARealAlleleSetArray alleles;
108
109   for (int i = 0; i < NUM_NODES; i++){
110     for (int j = 0; j < NODE_LENGTH - 1; j++)
111       alleles.add( 1, NUM_SENSOR_INPUTS + NUM_NODES, 1);//Node: All
              possible inputs.
112     alleles.add( 0, NUM_FUNCTIONS - 1, 1);//Last Element in Node:
          All possible functions.
113   }
114   for (int i = 0; i < NUM_GRAPH_OUTPUTS; i++)
115     alleles.add( NUM_SENSOR_INPUTS + 1, NUM_NODES +
          NUM_SENSOR_INPUTS, 1);//Output: All possible inputs.
116
117   //Data for constant nodes:
118   for (int i = 0; i < NUM_NODES; i++)
119     alleles.add( -100, 100, 1);//Will be scaled to between -1 and
          +1.
120
121   return alleles;
122 }
123
124
125 void printChromosome(int* printarray)
126 {
127   int nodeCounter = NUM_SENSOR_INPUTS + 1;
128   for(int i = 0; i < CHROMOSOME_LENGTH; i++)
129   {
130     if(i % 15 == 0) //formatting
131       cout << "\n";
132     if( i % NODE_LENGTH == 0) {
133       if(printarray[i] > 10) //formatting
134         cout << " ";
135       if (i >= (CHROMOSOME_LENGTH - NUM_GRAPH_OUTPUTS)) cout << "("
            << printarray[i] << ", ";
136       else {
137         cout << "[" << nodeCounter << "](" << printarray[i] << ", ";
              //beginning of node
138         nodeCounter++;
139       }
140     }
141     else if ( (i + 1) % NODE_LENGTH == 0)
142       if (i == (CHROMOSOME_LENGTH - 1))
143         cout << " " << printarray[i] << ") ";//Last elements in
              array are outputs.
144       else
145         cout << "_" << printarray[i] << ") ";//last element in node,
               function identifier.
146     else
147       cout << printarray[i] << ",";
148   }
149   cout << "\n"; cout.flush();
150 }
```

```
151
152 void printBestInd(GAGenome& g)
153 {
154   GARealGenome& genome = (GARealGenome&)g;
155   int nodeCounter = NUM_SENSOR_INPUTS + 1;
156   for(int i = 0; i < CHROMOSOME_LENGTH; i++)
157   {
158     if(i % 12 == 0) //formatting
159       cout << "\n";
160     if( i % NODE_LENGTH == 0) {
161       if (i >= (CHROMOSOME_LENGTH - NUM_GRAPH_OUTPUTS)) cout << "("
              << genome.gene(i) << ", ";
162       else {
163         cout << "[" << nodeCounter << "](" << genome.gene(i) << ", "
              ;//beginning of node
164         nodeCounter++;
165       }
166     }
167     else if ( (i + 1) % NODE_LENGTH == 0)
168       if (i == (CHROMOSOME_LENGTH - 1))
169         cout << " " << genome.gene(i) << ") ";//Last elements in
              array are outputs.
170       else
171         cout << "_" << genome.gene(i) << ") ";//last element in node
              , function identifier.
172     else
173       cout << genome.gene(i) << ",";
174   }
175   cout << "\n"; cout.flush();
176 }
177
178 #endif
```

Listing 2: Code/CGPglobals.h

```
1  #ifndef CGPGLOBALS_H
2  #define CGPGLOBALS_H
3
4  #include <ga/std_stream.h>
5
6  #define cin std::cin
7  #define cout STD_COUT
8  #define endl STD_ENDL
9
10 #define INSTANTIATE_REAL_GENOME
11 #include <ga/GARealGenomeED.h>
12
13
14 #define ROWS 5
15 #define COLUMNS 5
16 //#define ROWS 3
17 //#define COLUMNS 4
18 #define NODE_LENGTH 3
19 #define NUM_SENSOR_INPUTS 8
20 #define NUM_GRAPH_OUTPUTS 9
21
22 //Sensor identifiers, so we won't have to remember what number they
       are.
23 enum {CLOCK, TOUCH_CENTER, TOUCH_FRONT_LEFT, TOUCH_FRONT_RIGHT,
       TOUCH_BACK_LEFT, TOUCH_BACK_RIGHT, DISTANCE_LEFT, DISTANCE_RIGHT
       };
24
```

```
25 #define MAX_DISTANCE 100
26
27 #define NUM_NODES ROWS * COLUMNS
28 #define CHROMOSOME_LENGTH NUM_GRAPH_OUTPUTS + (NUM_NODES *
       NODE_LENGTH)
29 #define MAX_NUM_OUTPUTS NUM_SENSOR_INPUTS + NUM_NODES +
       NUM_GRAPH_OUTPUTS
30
31 #define MAX_ARITY 2
32 #define NUM_FUNCTIONS 20
33
34 double constant_array[NUM_NODES] = {0};//array to store constants
       for constant node function
35 double sine_wave = 0.5;
36 double sine_wave1 = 0.5;
37 double sine_wave2 = 0.0;
38
39
40 enum nodeFunctionGroupSet{FUNCSET_BASIC, FUNCSET_CONST,
       FUNCSET_COMPARE, FUNCSET_SINECOS, FUNCSET_LOG, FUNCSET_CONSTCOMP
       , FUNCSET_SINECOSCOMP1, FUNCSET_SINECOSCOMP2};
41 enum sensorEnableSet{SENSOR_NONE, SENSOR_TOUCH, SENSOR_DIST,
       SENSOR_JOINT};
42 nodeFunctionGroupSet nodeFunctionSet;
43 sensorEnableSet sensorSet;
44
45
46 #endif
```

Listing 3: Code/CGPnodefunctions.h

```
1 #ifndef CGPNODEFUNCTIONS_H
2 #define CGPNODEFUNCTIONS_H
3 //borrowed code from Millers CGP implementation.
4
5 #include <math.h>
6 #include "../cgp/CGPglobals.h"
7
8
9 double nodeFunction(double input[MAX_ARITY], int functionIdentifier,
       int node)
10 {
11   double result = 0.0;
12 if (nodeFunctionSet == FUNCSET_BASIC)
13 {
14   switch(functionIdentifier)
15   {
16   // addition
17     case 0: case 9: case 13:
18       result = input[0] + input[1];
19       break;
20   // subtraction
21     case 1: case 10: case 14:
22       result = input[0] - input[1];
23       break;
24   // multiplication
25     case 2: case 11: case 18:
26         result = input[0]*input[1];
27       break;
28   // protected division
29     case 3: case 12:
30       if (fabs(input[1]) < 0.0001)
```

```
31          result = input[0];
32        else
33          result = input[0]/input[1];
34        break;
35 // /*
36   // sine wave
37     case 6: case 15: case 19:
38       result = sine_wave;
39       break;
40 //*/
41 /*
42   // sine wave 1
43     case 6: case 19:
44       result = sine_wave1;
45       break;
46   // sine wave 2
47     case 15:
48       result = sine_wave2;
49       break;
50 //*/
51   // min
52     case 7: case 16: case 4:
53       result = __min(input[0],input[1]);
54       break;
55   // max
56     case 8: case 17: case 5:
57       result = __max(input[0],input[1]);
58       break;
59   }
60 }
61 if(nodeFunctionSet == FUNCSET_CONST)
62 {
63   switch(functionIdentifier)
64   {
65   // addition
66     case 0: case 9: case 13:
67       result = input[0] + input[1];
68       break;
69   // subtraction
70     case 1: case 10: case 14:
71       result = input[0] - input[1];
72       break;
73   // multiplication
74     case 2: case 11:
75         result = input[0]*input[1];
76       break;
77   // protected division
78     case 3: case 12:
79       if (fabs(input[1]) < 0.0001)
80         result = input[0];
81       else
82         result = input[0]/input[1];
83       break;
84   // evolved constant
85     case 5: case 4: case 18:
86       result = constant_array[node];
87       break;
88 // /*
89   // sine wave
90     case 6: case 15: case 19:
91       result = sine_wave;
92       break;
```

```
93 //*/
94 /*
95   // sine wave 1
96     case 6: case 19:
97       result = sine_wave1;
98       break;
99   // sine wave 2
100    case 15:
101      result = sine_wave2;
102      break;
103 //*/
104   // min
105    case 7: case 16:
106      result = __min(input[0],input[1]);
107      break;
108  // max
109    case 8: case 17:
110      result = __max(input[0],input[1]);
111      break;
112  }
113 }
114 if(nodeFunctionSet == FUNCSET_COMPARE)
115 {
116   switch(functionIdentifier)
117   {
118   // addition
119     case 0: case 9: case 13:
120       result = input[0] + input[1];
121       break;
122   // subtraction:
123     case 1: case 10: case 14:
124       result = input[0] - input[1];
125       break;
126   // multiplication
127     case 2: case 11:
128         result = input[0]*input[1];
129       break;
130   // protected division
131     case 3: case 12:
132       if (fabs(input[1]) < 0.0001)
133         result = input[0];
134       else
135         result = input[0]/input[1];
136       break;
137   // absolute
138     case 4: case 18:
139       result = fabs(input[0]);
140       break;
141   // compare
142     case 5: case 19:
143       if (input[0] < input[1])
144         result = -1.0;
145       else if (input[0] > input[1])
146         result = 1.0;
147       else
148         result = 0.0;
149       break;
150 // /*
151   // sine wave
152     case 6: case 15:
153       result = sine_wave;
154       break;
```

```
155 //*/
156 /*
157   // sine wave 1
158     case 6:
159       result = sine_wave1;
160       break;
161   // sine wave 2
162     case 15:
163       result = sine_wave2;
164       break;
165 //*/
166   // min
167     case 7: case 16:
168       result = __min(input[0],input[1]);
169       break;
170   // max
171     case 8: case 17:
172       result = __max(input[0],input[1]);
173       break;
174   }
175 }
176 if(nodeFunctionSet == FUNCSET_SINECOS)
177 {
178   switch(functionIdentifier)
179   {
180   // addition
181     case 0: case 9:
182       result = input[0] + input[1];
183       break;
184   // subtraction
185     case 1: case 10:
186       result = input[0] - input[1];
187       break;
188   // multiplication
189     case 2: case 11: case 18:
190         result = input[0]*input[1];
191       break;
192   // protected division
193     case 3: case 12:
194       if (fabs(input[1]) < 0.0001)
195         result = input[0];
196       else
197         result = input[0]/input[1];
198       break;
199   // sin (a+b)
200     case 4: case 13:
201       result = sin(input[0]+input[1]);
202       break;
203   // cos(a+b)
204     case 5: case 14:
205       result = cos(input[0]+input[1]);
206       break;
207 // /*
208   // sine wave
209     case 6: case 15: case 19:
210       result = sine_wave;
211       break;
212 //*/
213 /*
214   // sine wave 1
215     case 6: case 19:
216       result = sine_wave1;
```

```
217       break;
218   // sine wave 2
219     case 15:
220       result = sine_wave2;
221       break;
222 //*/
223   // min
224     case 7: case 16:
225       result = __min(input[0],input[1]);
226       break;
227   // max
228     case 8: case 17:
229       result = __max(input[0],input[1]);
230       break;
231   }
232 }
233 if(nodeFunctionSet == FUNCSET_LOG)
234 {
235     switch(functionIdentifier)
236   {
237   // addition
238     case 0: case 9:
239       result = input[0] + input[1];
240       break;
241   // subtraction
242     case 1: case 10:
243       result = input[0] - input[1];
244       break;
245   // multiplication
246     case 2: case 11:
247         result = input[0]*input[1];
248       break;
249   // protected division
250     case 3: case 12:
251       if (fabs(input[1]) < 0.0001)
252         result = input[0];
253       else
254         result = input[0]/input[1];
255       break;
256   // exp
257     case 18: case 19:
258       result = exp(input[0]);
259       break;
260   // protected natural log
261     case 4: case 13:
262       if (input[0] < 0.0001)
263         result = input[0];
264       else
265         result = log(fabs(input[0]));
266       break;
267   // protected log to base 10
268     case 5: case 14:
269       if (input[0] < 0.0001)
270         result = input[0];
271       else
272         result = log10(fabs(input[0]));
273       break;
274 // /*
275   // sine wave
276     case 6: case 15:
277       result = sine_wave;
278       break;
```

```
279 //*/
280 /*
281   // sine wave 1
282     case 6:
283       result = sine_wave1;
284       break;
285   // sine wave 2
286     case 15:
287       result = sine_wave2;
288       break;
289 //*/
290   // min
291     case 7: case 16:
292       result = __min(input[0],input[1]);
293       break;
294   // max
295     case 8: case 17:
296       result = __max(input[0],input[1]);
297       break;
298   }
299 }
```

Listing 4: Fitness scoring

```
1  float calculateSensorQuadrobotFitness(GAGenome& g)
2  {
3    SensorQuadrobotEvolutionHarness* evoHarness = (
         SensorQuadrobotEvolutionHarness*)g.userData();
4
5    if(!evoHarness->isRunning())
6      return 0;
7    if(gPhysicsSDK)
8      terminatePhysics();
9
10   float evalDuration=7.0f; //seconds
11   int runFrames=(int)(evalDuration/QUADRO_TIMESTEP);
12
13   float fitnessValue = 0.0;
14
15   if (!evoHarness->useGoals)
16   {
17     initPhysics();
18
19     gScene->setGravity(NxVec3(0,-9.81*100.0f,0)); //depending on
           scale
20
21     SensorQuadrobot machine = SensorQuadrobot(g);
22
23     NxVec3 firstPos=machine.getPosition();
24     NxVec3 lastPos=firstPos;
25     double distSum=0.0;
26     bool cheat=false; //for detecting simulator flaws or other
           unwanted behavior
27     double simTime=0;
28
29     evoHarness->obstaclesSet = false;
30
31     int f;
32     for(f=0; f<runFrames && !cheat; f++) {
33       bool freeze=false;
34       if(glfwGetKey(GLFW_KEY_F3)) freeze=true;
35       if( glfwGetKey(GLFW_KEY_ESC) ) {
```

```
36          evoHarness->stop();
37          break;
38        }
39
40        if(freeze && f>0) //freeze hack, ugly
41          f--;
42
43        machine.update((float)simTime);
44        setFollowTargetAndPanning(Vec3(lastPos.x,lastPos.y,lastPos.z)
              ,0);
45        evoHarness->updateGraphicsAndPhysics(freeze);
46        if(!freeze)
47          simTime+=evoHarness->m_timestep;
48        lastPos=machine.getPosition();
49
50    //Cheat sensors:
51        if(machine.m_SensorArray[TOUCH_CENTER]==true) {
52            cheat=true;
53            break;
54          }
55
56      } //end eval loop
57
58      NxVec3 diff=machine.getPosition()-firstPos;
59      terminatePhysics();
60      fitnessValue=(float)__max(diff.magnitude(),0); //allows any
            direction of final movement, not only "forward"
61      fitnessValue=fitnessValue/evalDuration; //should give cm/s
62
63      if(cheat) {
64        printf(" cheat!");
65        fitnessValue=0;
66      }
67      if (fitnessValue <= 3.692080 && fitnessValue >= 3.692078)
            fitnessValue -= 1.0;
68    }
69
70    printf("\t\t\tfit: %f\n",fitnessValue);
71
72    return fitnessValue;
73  }
```